MPE Forth Cross Compiler

versions 4.1/4.1H

User Manual

December 1988
Last Revision - February 1990

MPE/Nautilus Forth Cross Compiler

versions 4.1/4.1H

User Manual

MicroProcessor Engineering Ltd
133 Hill Lane
Shirley
Southampton S01 5AF
England
tel: (+44) 703 631441
fax: (+44) 703 339691

by

Jonathan Lee

and

Stephen Pelc

Table of Contents

Introduction

The MPE Forth Cross Compilers generate Forth-83 object code for a very wide range of processors, ranging from 4-bit single chip processors used in washing machines, to 32-bit processors such as the Transputer. The cross compiler is particularly optimised for generating embedded systems, but the same ersion can be, and is, also used for porting Forth across different operating systems. This compiler has been generated by engineers for engineers.

The supplied target source code is an MPE implementation to the Forth-83 standard. This implementation has many extensions to the Forth-83 standard, and is called PowerForth. Apart from functions specific to particular processors or operating systems, versions of PowerForth are source code compatible with each other. Included with the source code is an interrupt handler, and multi-taskers and other extensions are optionally available.

The process of installing Forth on a target system is quite straightforward. The cross compiler runs on the host computer, and generates an object file which is then used on the target. The target source code is edited to define the memory usage, and to define the serial interface used to connect the host to the target. The transfer of the object code is performed by programming an EPROM, using an EPROM emulator such as the Leburg LePROM, or by downloading via a monitor program. Once the target code is running, the target Forth can now be extended.

Communications software is provided to talk to the target in the form of the T-COMM utility, the host-end of which is supplied preinstalled into the cross compiler support shell. The support shell - XShell and PowerForth are also supplied and documented separately. This code is provided on a separate disc, and is documented in another section of the manual.

Installation of the software is covered in a separate chapter of this manual.

This compiler is a very powerful tool. It is worth spending some time learning to use it. To this end you should be confident with Forth itself before you start. Confidence in Forth itself is best gained by writing an application on the host system. This will teach you about the host's editor, operating system, and utilities. A suggested initial task is to write a utility to download files to another computer or an EPROM programmer. You can use the tools in XC-COMM.SCR as an example. Once you are confident in Forth start using the cross compiler.

The manual for the PowerForth system suppled with the cross compiler includes a tutorial chapter. Also recommended are Leo Brodie's books 'Starting Forth' and 'Thinking Forth'.

1.1 What the cross compiler does

A Forth cross compiler is a tool for generating a new Forth system. The compiler itself is written in Forth and runs on a computer which is called the host. The new Forth system may be for the same or different hardware, or even for a different processor altogether. The new Forth system is called the target. The output of the cross compiler is placed in a disc file on the host computer. It may then be transferred to the target by serial loading, blowing an EPROM, or by other means.

To allow the cross compiler to cope with these changes it is written in two parts.

The first part, called the core, deals with the mechanics of compiling, forward references, and so on. This manual documents the core, which is common to all the compilers, regardless of the target processor. The second part is the cross assembler, which copes with the different instruction sets of the different processors. When a cross compiler for a new processor is required, a new cross assembler is written, and is then combined with the core to produce a cross compiler for that processor. The assembler sections of the source code are then rewritten for the new processor. The cross assembler and target are documented in the target manuals.

The cross compiler acts upon the target source code, which represents the complete source code for a particular Forth implementation. The source code looks like normal Forth, and indeed it is normal Forth source code.

The cross compiler transforms this source code into a memory image, which is stored on the host's disc. The image is a complete program, which can be executed when transferred to the target hardware.

The supplied target code is an executable Forth, which compiles and interprets like any other Forth. The process of extending the supplied Forth to cope with the required application is very similar to developing a normal Forth application. New source code is added to the target nucleus, compiled and tested, and finally the application emerges.

What is special about the cross compiler is that it contains many features, such as forward referencing, that are not normally permitted in Forth systems. These features are tuned to generating sealed application programs. Because

the user has complete source code of the target code, the user can tune the application to his needs very finely.

The cross compiler is designed to be used without a deep understanding of the compiler itself. It is also designed to process source code as close as possible to normal Forth source code. We hope that you will find the additional information useful as your experience grows. Full source code of both the compiler and the target code is supplied so that you can extend the power of the system as far as you need.

1.2 Sequence of operations

The sequence of operations is normally Edit, Cross compile, Transfer, and Debug. Unlike many other cross support systems, debugging a Forth system is performed interactively, avoiding the time lost continually recompiling code with yet another print function included. The system is optimised for rapidity of job turn round, and each component of this cycle can be individually optimised. The XSHELL2 support package allows the user to select the correct tool with a single key stroke.

1.2.1 Editing

The cross compiler accepts source code from two types of file. These may be text files as produced by any conventional program editor, or may be the more traditional Forth screen files. XSHELL2 supports either, as does the compiler, and it is even possible to work in a mixed environment, with part of the application using text files and part using screen files.

The Forth screen editor is accessed from the **<ALT> F** key. If you prefer to use normal text files this is available from the **<ALT> A** key. To start, you will use the editor to set up the target memory definitions, and to modify the UART drivers so that the target can communicate with the host. Once the target debugging phase has started the editor is used to install TESTED corrections into the target source code. Most of the changes will have been proven by the interactive debugging.

Details of the editor and editing are to be found in the editing chapter of this manual, and in the host PowerForth manual.

1.2.2 Cross Compiling

Once the target source code has been modified for the specific hardware and memory you are using, run the cross compiler from the **<ALT> C** key. When the compiler signs on, a phrase of the form:

2 LOAD

for screen files, or of the form:

2 ONWARDS FROM

for text files, will run the compiler to completion or until an error occurs. If an error occurs go back to the editor to correct it. The compiler will tell you where the error occurred. A successful compilation will leave an image file, ROMxxx.IMG, on the disc. If your computer supports a RAM disc, you can save some time by editing ROMCTL.SCR to place the output file on the RAM disc using the **OUTPUT-FILE** directive, for example:

OUTPUT-FILE E:ROM6811.IMG

More time can be saved using the **NO-LOG** directive to turn off the display of the output map:

NO-LOG

Using **OUTPUT-EMULATOR**, instead of the **OUTPUT-FILE**, directs the compiler to put the binary code straight into the Leburg LEPROM EPROM emulator instead of a file on disc. This saves even the time taken to transfer the image to the target.

There is a later chapter in this manual on ‘Controlling the Compiler’, and the ‘Compiler Directives Glossary’ with reference entries for each directive.

There is a later chapter in this manual on ‘Controlling the Compiler’, and the ‘Compiler Directives Glossary’ with reference entries for each directive.

1.2.3 Transferring the Image

The cross compiler produces a binary image of the target system. This means that each byte in the output file represents one byte in the target image. This simplest of formats can be used by most EPROM programmers and emulators that plug directly into a PC. Serially connected devices use a variety of formats, the most common of which are Intel Hex and Motorola S-record. Both these formats are supported by the XC-COMM tools. See the XC-COMM manual for further details.

From the point of view of turn round time, a PC plug-in EPROM emulator such as the Leburg LePROM is far and way the fastest means of getting code into a target system. The next best option is a PC plug-in EPROM programmer

such as the Sunshine EW-701 supplied by MPE. Both these solutions avoid the time taken to transfer data across a serial link.

1.2.4 Talking to the target

The usual means of talking to the target is by the serial line on the PC connected to an RS232 port on the target. The target serial port can be from a UART on the target processor chip, an external UART, or a simulated 'bit--banging' software UART. Anything that works is reasonable. Other techniques that have been used are SCSI ports, bi-directional parallel ports, and dual-port RAM.

The built-in comms package can be accessed by pressing **<ALT> T** or **<ALT> S** in XShell. Note that the function keys and ALT keys are still available while Terminal mode is active. The default baud rate is 9600 baud, or whichever rate was set when the XShell configuration was saved. See the XShell v2 manual for further details, and the XC-COMM manual for hints and tips on installing a serial link.

1.2.5 Debugging the target

Debugging target code falls into one of two classes, bringing up Forth for the first time, and debugging the application. Bringing up Forth for the first time is a matter of getting the serial line working, and then **EMIT**ting characters to check progress. Once the target Forth interpreter/compiler is running, the target can be debugged in the same way as a Forth system on the host computer. The T-COMM utilities includes code for downloading screen files from the host and XShell, and any conventional comms package can be used for downloading text files. Corrections should be tested and then added to the source code as you go along.

Debugging is discussed in a later chapter of this manual.

1.2.6 Autorunning applications

The final application does not need to be interactive. Once the application is debugged the additional debugging code, and the Forth interpreter/compiler layer, can be removed if desired. The cross compiler is instructed by the directive **NO-HEADS** not to include any dictionary heads in the target at all. The startup code of the target is modified in **COLD** or **ABORT** to fall into the application loop rather than the interactive loop. This code is then cross compiled one last time, and a final EPROM set programmed.

It is worthy of note that many of our users leave the dictionary heads and Forth interactive layer in the final target code, making it accessible as an engineering function.

Source Code and Editing

2.1 Editing Styles

Forth source code can be handled in two ways. The first way is to use conventional text files as produced by any standard program editor. The XShell2 environment can be set up to call any standard editor using a single keystroke. The system used to handle text files is discussed later in this chapter. The second way is to use Forth screen files. The organisation of these files is a result of the limited hardware available when Forth was first implemented. If source code has to be transferred to a stand-alone computer without a filing system, screen files are still a good way to work.

There is an increasing use of conventional text files by the Forth community, but because of the quantity of code already written in screen file format the two formats will have to coexist for some time to come. Because of its suitability for embedded and minimal systems, the screen file format will remain with us for a long time.

The cross compiler is designed to work equally well with either type of file, and in any application either one or both types of file may be freely used.

2.2 Using conventional files and editors

The cross compiler can compile source code from any standard ASCII file generated by a standard program editor. Word processors that leave control codes or use the top bit for control purposes should be avoided.

In order to maintain the degree of selective loading available with screens, code can be compiled from pages in a file. The use of pages for Forth source code restores all the features normally attributed to screen files. Code can be selectively loaded, restoring the use of source code libraries. Pages are numbered from one onwards. A page is denoted by ASCII code 12 (0Ch), often generated by ^L. Editors which support paging, such as the WordPerfect Program Editor, Brief, or MicroEMACS, are strongly recommended.

2.2.1 Configuration

Within Xshell2 the editor to be used and the current text file can be defined:

```
EDITOR-IS <pathname>          \ define full path to editor
USE <pathname>   \ define full path to text file
```

The editor is then called by:

```
ED [<pathname>]
```

If the pathname is not supplied, the current text file will be used, otherwise the given file will be edited.

From the XShell environment, the **<ALT> A** key invokes the editor. The configuration system on the **<ALT> K, b** keys allows the editor to be defined by the user.

2.2.2 Compiling from Text files

From within the XShell2 environment, the cross compiler is loaded and run using the **<ALT> C** key, which can be configured by the configuration from the **<ALT> K,b** keys. The default configuration supplied by MPE also calls the compiler using the **F3** key.

From within Xshell2 or from the cross compiler, a text file can be loaded by using the phrase:

```
n1 n2 FROM-FILE <path-name>  \ from any file
n1 n2 FROM                                  \ from current file
```

where n1 is the starting page and n2 is the last page to be compiled. A set of modifiers is available to generate ranges easily.

```
n1 ALONE                    \ n1 — n1 n1
n1 ONWARDS                  \ n1 — n1 -1
ALL                                 \ — 1 -1
```

Thus to compile everything from the file DRIVERS.4TH use the phrase:

```
ALL FROM-FILE DRIVERS.FTH
```

If the file has no page breaks use one of the styles shown below. The use of page breaks is not essential, it simply allows a natural format to be used without introducing printer directives. A major benefit of page breaks is to retain the facility of compiling parts of a file during interactive testing on the host. Without such a mechanism, code is written in a vast number of very small files, transferring the problem area to that of file management on the disc. The page mechanism allows the user to choose the size of text file, while retaining convenient partitions.

```
ALL FROM
```

or

```
1 ONWARDS FROM
```

or

```
1 ALONE FROM
```

2.2.3 Text File words Glossary

Listed below are a number of words to enable text file input. Along with their names is shown, pronunciation, stack effect, input stream effect, a description, and any similarity to the Forth block loading words.

USE — ; <text-file-name>
"use"

Set the default text file you wish to **USE** (like "**USING** xxx.scr" for screen files). The file defaults to FORTH.FTH.

.FTH —
"dot-forth"

Print the default text file (like **.SCR**).

EDITOR-IS — ; <editor-file-name>
"editor-is"

Set the file-name of the editor you wish to use (originally set to "C:\EDT\EDT.EXE").

ED — ; <text-file-name>
"ed"

Call the EDitor, using either the default file or a file-name typed after **ED**.

ED ANYFILE

FROM first last —
"from"

Get text input from a range of pages in the current file. The first and last pages to be used are supplied on the stack. Files can be nested to any depth; that is files can include input from other files. **FROM** is like **THRU** or **THRU-USING** with screen files, e.g.

4 10 FROM

FROM-FILE first last — ; <filename>
"from-file"

Get text input from a range of pages in the file <file-name> typed after **FROM-FILE**. The first and last pages to be used are supplied on the stack. Files can be nested; that is files can include input from other files. **FROM-FILE** is like **THRU** or **THRU-USING**, e.g.

4 10 FROM-FILE MYFILE.FTH

DISPLAY first last — ; <text-file-name>
"display"

Display the text of a range of pages in the file. The file is either the file-name typed after **DISPLAY**, or the default file. The first and last pages are supplied on the stack. (Similar to **SHOW**).

CONTENTS first last — ; <text-file-name>
"contents"

Display the contents of a range of pages in the file, that is display the comment (first) line from each page. The file is either the file-name typed after **CONTENTS**, or the default file. The first and last pages are supplied on the stack. Like **INDEX** for screen files.

QC — ; <text-file-name>
"q-c"

Print the CONTENTS of the whole file. The file is either the file-name typed after **QC**, or the default file. (Like **QX** for screen files.)

ALL — 1 -1
"all"

Used before **FROM**, **DISPLAY**, or **CONTENTS**, **ALL** will put the first and last page numbers on the stack so that ALL the pages are specified. e.g.

```
ALL DISPLAY MYFILE.FTH
```

ALONE n — n n
"alone"

A modifier used before **FROM** or **FROM-FILE** or other text file words to describe a single page, for example to compile page 5 from the current text file use:

```
5 ALONE FROM
```

ONWARDS n — n -1
"onwards"

A modifier used before **FROM** or **FROM-FILE** or other text file words to use the text from a specified page to the end, for example to compile page 5 to the end from the current text file use:

```
5 ONWARDS FROM
```

.WHERE —
"dot-where"

Prints WHERE an error occured when loading from a file.

PTO —
"p-t-o"

Please Turn Over. This word causes **FROM** and **FROM-FILE** to stop using the current page and to start on the next. (The same effect as —> in screen files.)

;P —
"semi-p"

End Page. Causes **FROM** and **FROM-FILE** to stop using the current page. Has the same effect as **;S** in screen files.

(—
"paren"

The comment in a text differs from the screen file version. The paranthesis comment works over multiple lines. The end of comment is marked by a white-space-delimited closing parenthesis, e.g.

```
... 2DUP ( save adr # for later )  INIT ...
    ...
CLOBBER  ( NOTE:
                    use the operating system function
                    here to shut off access
         )
    ...
```

2.2.4 Examples of Text File Use

EDITOR-IS C:\EDT\EDT.EXE
USE MAIN
.FTH
Forth prints:

Default source file: MAIN.FTH ok

QC
Contents of MAIN.FTH are printed on the display screen (ie comment line of every page).

4 6 DISPLAY
pages 4, 5 & 6 displayed on screen.

ED
Edit MAIN.FTH using EDT.
Exit EDT.

4 6 FROM

Pages 4, 5 & 6 loaded.

2.3 Using Screen Files

2.3.1 Screen Files

The traditional unit of Forth source code is the 'screen', a 1024 byte record normally displayed as 16 lines of 64 characters each. Screens of source text are stored on disc in screen files, file size being limited by the available space on the disc. The screens are numbered from zero upwards, and are stored as randomly accessed 1024 byte records within a standard operating system file.

To allow conversion between screen files and text files tihe Forth utilities **FILE-EXPAND** and **FILE-COMPRESS** allow you to 'import' and 'export' files written using a standard text editor. The **FILE-COMPRESS** utility converts a screen file to a text file, and the **FILE-EXPAND** utility will convert a compressed file back to a screen file. These utilities can be found in the file FORTH.SCR supplied with PC PowerForth.

To open or change the default screen file being accessed, Forth provides the word **USING**. At startup the file FORTH.SCR will be used if available and you did not specify a different file on the command line. You can change the default file to (say) GAME.SCR by typing:

USING GAME

To return to the original screen file, just type:

USING FORTH

The word **USING** adds the extension .SCR to the file name if no other extension is specified. The word **BLOCK** is used by the editor to access blocks (screens) in the screen file, via Forth's internal buffers. The words **LOAD** and **THRU** are used outside the editor to interpret screens from the file you have been editing, e.g.:

```
6 LOAD          \ just load screen 6
7 15 THRU       \ load screens 7 to 15
```

Screens can be compiled from another file with **LOAD-USING** and **THRU-USING**. After the code has been compiled the original screen file will be restored. Only one level of nesting is permitted, which is sufficient to allow a control file (such as ROMCTL.SCR) to load all the other required screens from required files.

2.3.2 Introducing FRED

The screen file editor supplied in XShell2 is FRED, an editor specifically designed to edit Forth screen files. If you prefer to edit using a conventional text editor, a later section of this chapter describes and discusses the techniques available to you.

This editor has several features new to Forth editors, and is completely reconfigurable by the user (see the XShell2 manual for details). Traditionally, Forth has edited source files in terms of 1024 byte blocks, presented as 16 lines of 64 characters. These are known as screens, and are stored on disc in random access files under the host operating system. We have remained with this standard, but have added to it full screen editing, a line stack/barrel, as well as many other features. In the interests of producing a powerful editor that is easy to use, we have made no attempt whatsoever towards compatibility with the editor documented in Leo Brodie's otherwise excellent book 'Starting Forth'.

2.3.3 Entering the editor

Normally the editor will be called using the **<ALT> F** key within XShell v2. However, if you are working from the host's Forth command line, two Forth words are available. To start the editor use the words **EDIT** or **FRED**.

EDIT

EDIT will load the editor from disc and enter the editor, displaying screen 0, or the last screen previously edited.

The editor can be entered displaying a specific screen number by giving the screen number to the word **FRED**.

25 FRED

2.3.4 Screen File Glossary

LOAD n —
"load"

Loads and compiles the code on the given screen. Do not try to **LOAD** screen 0, although it can be **EDIT**ed and **LIST**ed. By tradition the screen being loaded is held in the variable **BLK**. If this variable is 0, input is taken from the keyboard. The range of screen numbers should be 1..32767.

```
5 LOAD          \ loads screen 5 from current screen file.
```

LOAD-USING n — ; n LOAD-USING <file-name>
"load-using"

As **LOAD**, but a file name is specified, and the screen is taken from that file. **LOAD-USING** can be called from within another file, but cannot be nested again. If **LOAD-USING** is called from another screen file, that file is made the current screen file again when the action of **LOAD-USING** has finished.

THRU first last —
"thru"

Performs the same function as **LOAD**, but for a range of screens.

THRU-USING n — ; n THRU-USING <file-name>
"thru-using"

As **THRU**, but a file name is specified, and the screen is taken from that file. **THRU-USING** can be called from within another file, but cannot be nested again. If **THRU-USING** is called from another screen file, that file is made the current screen file again when the action of **THRU-USING** has finished.

Blank Page

Controlling the Cross Compiler

3.1 Sequence of Operations

Using the cross compiler is in practice simple.

Firstly, set up a control file that tells the cross compiler what to do. The supplied control file ROMCTL.SCR is an example. This file tells the cross compiler what to compile from which files, and cleans up at the end. The target source code may also contain one or more files named xxxxx.CTL for specific configurations.

Secondly, define the memory layout of the target. This is done by compiler directives, usually in ROMCTL.SCR.

Thirdly, define the hardware terminal drivers. You will need to talk to the hardware, and the words **KEY? KEY** and **EMIT** provide the basic terminal console functions. If discs or mass storage are required, the words **BLK-READ** and **BLK-WRITE** used by **BLOCK** will need to be provided. The target files include example i/o code files, usually called ROM-IO.SCR.

Fourthly, compile it. Compiling is initiated by loading the master screen of the control file. Cross compilation starts when the word **CROSS-COMPILE** is encountered. From now on the cross compiler is enabled, and all code will be treated as target code. When the word **FINIS** is encountered, the cross compiler is turned off, final reports made, files closed, and the compiler terminates. If the compiler is not told otherwise the output will be in a file called ROMxxx.IMG on the default disc/directory.

Fifth, transfer the image to the target for testing. There are tools for this in the XC-COMM utility supplied with the compiler, including an Intel hex format utility as required by many EPROM programmers.

3.2 Setting up a control file

The control file selects which screens are to be used from which files. It can also be used to load additional assembler macros and compiler extensions before cross compiling starts. See later sections for details. The compiler directives most often used are shown here. These directives are unly used once, and apart from starting and stopping the compiler, they serve to describe the target processor to the compiler.

CROSS-COMPILE —
"cross compile"

This word tells the system to start cross compiling. Until this point the compiler is a conventional Forth system with an application (the cross compiler) loaded but not running. At this stage extensions and assembler macros can be loaded. After **CROSS-COMPILE** has executed the compiler 'pulls down the shutters' to seal itself off, and then treats all code as target code and compiler directives.

FINIS —
"finis"

This word stops the cross compiler. A cleaning-up operation is performed, final reports issued, the output file closed, and finally the compiler exits to the operating system.

ALIGN —
"align"

Used if the target processor requires 16-bit items to be aligned on a word boundary. This is sometimes a requirement of 16-bit targets such as the 68000 or 8096. This directive forces the cross compiler to align the link field and code field on even addresses. Some processors (such as the 8096) need the CFA to be aligned on an odd address. In this case use **ALIGN-ODD**.

ALIGN-EVEN —
"align-even"

A synonym for **ALIGN**.

ALIGN-ODD —
"align-odd"

Used for processors such as the 8096 family which need the CFAs to be on odd addresses (so that PFAs are on even ones). The link fields are still forced to a even address.

3.3 Defining memory layout

The cross compiler needs to know where in memory the output is to placed. **ROM-BASE** directive tells the cross compiler that the ROM area starts at the address given. The start of the output file will correspond to this address. The start of RAM is defined by the **RAM-BASE** directive, and the end of RAM is defined by the **RAM-END** directive.

The following words are cross compiler directives that control the memory organisation of the target.

MEM-BASE addr —
"mem-base"

Used to set the base address of memory for a RAM system, e.g.

```
HEX
0100 MEM-BASE
```

MEM-END addr —
"mem-end"

Used to set the upper address + 1 of available RAM for both RAM and ROM/RAM systems. Even if the target determines its own RAM allocation this directive should be used, so that the compiler error checking does not use silly limits. e.g. for a CP/M system whose executable file starts at 0100h:

```
HEX
0100 MEM-BASE
0E000 MEM-END
```

For a ROM-based system with ROM at 0000, and RAM from 08000 to 09FFF, the following sequence would be used:

```
HEX
0 ROM-BASE
08000 RAM-BASE
0A000 MEM-END
```

RAM-BASE addr —
"ram-base"

Used to set the base address of the RAM in a ROM/RAM system, e.g.

```
HEX
00000 ROM-BASE
08000 RAM-BASE
0A000 MEM-END
```

RAM-END addr —
“ram-end”

A synonym for **MEM-END**.

ROM-BASE addr —
“rom-base”

Used to give the base address of the ROM and informs the compiler that a ROM/RAM system is to be generated, e.g.

```
HEX
0500 ROM-BASE
```

3.4 Accessing target memory

The target meory image can be accessed while the cross compiler is running. Thus variables can have initial values stored into them and read later on. The code fragment below is valid, and will execute as it would on the host.

```
VARIABLE POINTER          \ declare a variable
 0F000 POINTER !          \ intiialise it

POINTER @ U.              \ show its value
```

The majority of the Forth memory operators have been modified to operate in this way. The target dictionay pointer is available as **HERE**, and the next free location in RAM is given by **THERE**, which acts as an analogue of **HERE**, but for RAM.

3.5 Output Control

The cross compiler defaults to using an output file called POWER.IMG if no other name is specified. The compiler directive **OUTPUT-FILE** is used to define the output file. If no extension is specified the default extension '.IMG' will be added.

```
OUTPUT-FILE <path-name>
OUTPUT-FILE ROM6811
                                    \ use file ROM6811.IMG
OUTPUT-FILE C:\IMAGES\CPMZ80.COM
                                    \ use output file CPMZ80.COM
```

The **OUTPUT-FILE** directive should be used before the **ROM-BASE** or **MEM-BASE** directives, as the output file is used once its memory base address has been defined. A separate set of words is used to control the output file for paged compilation, which is discussed in detail in a later section of this chapter.

OUTPUT-FILE — ; <pathname>
"output-file"

Sets the output file to be used for the target image. If no extension to the filename is given, the extension '.IMG' will be added. The file is not opened/created until the base address of the output has been set by **MEM-BASE** or **ROM-BASE**. Use in the form:

```
OUTPUT-FILE <pathname>
OUTPUT-FILE ROMZ80
```

If a Leburg LePROM EPROM emulator is available the cross compiler can direct output to it instead of to an output file. Note that you cannot have both together. The cross compiler will need to know the I/O port address where the emulator is installed in the PC, what sort of EPROM is being emulated, and what bus width is being emulated. The example below shows how this is done.

```
<i/o address> EMU-BASE
<EPROM type> <bus-width> OUTPUT-EMULATOR
```

where <i/o address> is a port number (usually hex 0300 for an XT, or hex 0320 for an AT), <EPROM-type> is one of:

E2764 27128 E27256 E27512

and <bus-width> is one of:

8BIT 16BIT 32BIT

To set up output to an emulator at address 0320h, using two 27128s as a 16 bit pair, use the following code:

```
HEX
0320 EMU-BASE
E27128 16BIT OUTPUT-EMULATOR
```

EMU-BASE address —
"emu-base"

Sets the base address of the LePROM emulator for the cross compiler.

OUTPUT-EMULATOR eprom-type bus-width —
"output-emulator"

Defines the EPROM type and bus width of the EPROM emulator the compiler will use. This directive switches all target memory words to use the EPROM emulator drivers instead of the output file drivers. Predefined constants exist to define the EPROM type and bus width. The EPROM type is one of:

E2764 27128 E27256 E27512

and the bus-width is one of:

8BIT 16BIT 32BIT

3.6 Input File Control

Source code may be loaded from text files or Forth screen files. It is not necessary to stick to one type only, as screen files may be **LOAD**ed from text files, or text files **FROM** screen files.

3.6.1 Compiling from Text files

The control file is selected from the command line with the **USE** <pathname> directive, followed a compile command. The following two examples are equivalent.

```
X6811 USE ROMCTL.4TH  2 ONWARDS FROM
X6811 2 ONWARDS FROM-FILE ROMCTL.4TH
```

The cross compiler control file is loaded as a normal Forth application. The normal PowerForth text file control words **FROM** and **FROM-FILE**, and their modifiers **ALL**, **ALONE**, and **ONWARDS** are available and will act as expected. Text files may be separated into pages (^L, ASCII code 12/0Ch) and the range of pages can be specified. This model is more fully described in the chapter on source code and editing. If your editor does not support pages, use the **ALL** modifier with **FROM** and **FROM-FILE**.

Nesting of text files is supported, the level of nesting being limited by the PC's memory. In pratice, this allows for far more levels than are likely to be needed.

USE — ; <text-file-name>
"use"

Set the default text file you wish to **USE** (like "**USING** xxx.scr" for screen files). The file defaults to FORTH.4TH.

FROM first last —
Get text input from a range of pages in the current file. The first and last pages to be used are supplied on the stack. Files can be nested to any depth; that is files can include input from other files. **FROM** is like **THRU** or **THRU-USING** with screen files, e.g.

```
4 10 FROM
```

FROM-FILE first last — ; <filename>
Get text input from a range of pages in the file <file-name> typed after **FROM-FILE**. The first and last pages to be used are supplied on the stack. Files

can be nested; that is files can include input from other files. **FROM-FILE** is like **THRU** or **THRU-USING**, e.g.

```
4 10 FROM-FILE MYFILE.4TH
```

ALL — 1 -1

Used before **FROM**, **DISPLAY**, or **CONTENTS**, **ALL** will put the first and last page numbers on the stack so that ALL the pages are specified. e.g.

```
ALL DISPLAY MYFILE.4TH
```

PTO —

Please Turn Over. This word causes **FROM** and **FROM-FILE** to stop using the current page and to start on the next. (The same effect as **—>** in screen files.)

;P —

End Page. Causes **FROM** and **FROM-FILE** to stop using the current page. Has the same effect as **;S** in screen files.

(—

The comment in a text differs from the screen file version. The paranthesis comment works over multiple lines. The end of comment is marked by a white-space-delimited closing parenthesis, e.g.

```
... 2DUP ( save adr # for later )  INIT ...
   ...
CLOBBER  ( NOTE:
                    use the operating system function
                    here to shut off access
    )
   ...
```

3.6.2 Compiling from Screen files

The words used to compile from screen files specify a single screen or a range of screens. These words work with the cross compiler in the same way as they do on the host PowerForth.

LOAD n —
"load"

The contents of screen n of the current screen file are compiled, e.g.

```
10 LOAD
```

LOAD-USING n — ; <pathname>
"load-using"

The contents of screen n of the given screen file are compiled. If no extension is given, the compiler will add the extension '.SCR', e.g.

```
10 LOAD-USING A:\ROM-IO     \ load from A:\ROM-IO.SCR
```

THRU n1 n2 —
"through"

The contents of screens n1 to n2 inclusive of the current screen file are compiled, e.g.

```
DECIMAL 7 23 THRU
```

It is good practice to define the number base just before **LOAD** or **THRU** as 'other people' sometimes forget to restore the base at the end of a screen, or it might be house policy only to define the base where it matters. It this instance it does.

THRU-USING n1 n2 — ; <pathname>
"through-using"

The contents of screens n1 to n2 of the given screen file are compiled. If no extension is given, the compiler will add the extension '.SCR', e.g.

```
DECIMAL 2 75 THRU-USING WRK83          \ from WRK83.SCR
```

3.7 Terminal drivers

The usual way to talk to the target hardware is via a serial link connected to the host computer. It is thus necessary to write some simple routines to test whether input is ready, input a character, and write a character. These routines correspond to the Forth words **KEY? KEY** and **EMIT**.

KEY? tests the serial line and returns true if a character has been received, or false if no character has been received.

KEY? — t/f

KEY waits until a character has been received on the serial line, and then returns its value.

KEY — char

EMIT sends a character out on the serial line, waiting if necessary until the previous character has been sent.

From the point of view of getting the system going, ANYTHING that works is acceptable. You can always rewrite it later. The important thing is to get something working. Once the serial line is working you can quickly insert test code to help debug the application. Once **EMIT** in particular is working using can insert phrases of the form:

ASCII A EMIT

into your code so that you can display what the program has executed.

You will find sample code for these words in the target i/o handler file, which is usually called ROM-IO.SCR. This file contains example implementations that we have used ourselves. Edit the code for your particular hardware. Note that in practice nearly all UARTS work in the same way. A status bit has to be polled until the device is ready. The status bit is usually in a status register and has to be ANDed with a suitable mask value to extract it. When the device comes ready, a register can be read or written. This is usually sufficient, but on some devices, another register or bit has to be manipulated to reset the status bit that was being polled. The complexities are usually caused by the initialisation code. Note that the word **INIT-SER** (see ROM-IO.SCR) should contain all the UART initialisation code.

3.8 Headerless code

In an application program, it is often unnecessary to be able to access some or all of the Forth words by name. A sealed program can save space by not including the Forth words that deal with text interpreting and compiling, and so the ASCII text of the word names need not be included. This is called headerless code. Other application programs that use interpretation as, say, a command scanner, can save memory space by removing the heads from all words except those that form the command vocabulary. Commercial applications running under a host operating system sometimes use also headerless code for application security, as it inhibits reverse engineering.

The cross compiler supports generation of headerless code, and directives are available to switch generation of heads on or off on the fly. It is also possible to restrict the name field width to a maximum number of characters. This possibility allows you to save space at the risk of name conflicts.

The following compiler directives control generation of headerless code.

EXTERNAL —
"external"

The following words are generated with headers containing up to 31 characters, provided that the directive **NO-HEADS** has not been used.

```
EXTERNAL
```

HEADS? — t/f I
"heads-query"

An immediate equate that returns false if the **NO-HEADS** directive has been used. The function of this equate is to return a value for condition compilation of the interpreter and compiler layers if heads are needed, for example:

```
HEADS?
IF( 7 LOAD-USING INTERACTIVE )ENDIF
```

INTERNAL —
"internal"

This directive causes the following words to be generated without headers. The cross compiler still knows they are there, but they will not be visible to the compiled system.

INTERNAL

NO-HEADS —
"no-heads"

Disables generation of heads, overriding **EXTERNAL** and **TARGET-WIDTH** completely. This directive can be used when the application is complete to remove ALL heads from the system without having to go through the source code removing all occurences of **EXTERNAL** and **TARGET-WIDTH**.

NO-HEADS

TARGET-WIDTH n —
"target-width"

Sets the maximum number of characters in the name field to be n. A maximum of 31 characters is imposed by the compiler.

7 TARGET-WIDTH

3.9 Labels and Equates

It is often useful to be able to define constants which do not appear in the target, but sinply return numbers. This facility is equivalent to the equate system found in most assemblers. The rules for the use of labels and equates are straightforward.

3.9.1 Equates

An equate is generated by the word **EQU** using the syntax:

n EQU <name>
5 EQU IO-PORT

Equates cannot be forward referenced, as any forward reference when compiling refers to a Forth word, and when assembling a **LABEL** is referred to. A reference to to an equate returns its value during interpretation or assembly, and compiles a literal of that value during compilation.

Equates may be redefined at any time, so that they may be used like 'set--symbols' in a conventional assembler. The compiler will issue a message at every redefinition.

3.9.2 Labels

A label refers to a location in either ROM or RAM. Labels are defined by the words **L:** (“ell-colon”) using the syntax:

L: <name>
L: MARKER

L: is used to generate a label at the current location, which will be the current value of **HERE**. During assembly, a forward reference is assumed to be to a label, so allowing jump or branch targets to be forward referenced. If the forward reference is to a data location, it must be resolved by the declaration of a label.

To this end, the directive **LABEL** is used to define a label with an arbitrary value. Its main function is to allow code fragments to be generated in RAM so that they can be modified. This ugly practice is necessary to obtain efficient code in some processors. For example, code can stored in battery backed RAM, which does not form part of the memory allocated by **RAM-BASE** and **RAM-END**.

<addr> LABEL <name>

0F000 LABEL RTC-RAM

3.10 **Conditional Compilation**

If a piece of target code has to be used in different environments, say on different processors, and thus has to have two versions, it is useful to be able to compile it differently according to the required version. This is quite easy if the conditions can be expressed only during interpretation. Generate an equate to be used a flag and use the words **IF( )ELSE(** and **)ENDIF**. These are analogues of the normal **IF ELSE** and **ENDIF**. The words after **IF(** are dealt with if the flag is true (non-zero).

```
1 EQU DO-VERSION-1          \ compile version-1 when set

DO-VERSION-1 IF(
: VERSION-1       ..... ;
)ELSE(
: VERSION-2       ..... ;
)ENDIF
```

To use conditional compilation during cross compilation is a little more complicated as the flag must be determined while the cross compiler is compiling. To make this happen the flag must be added to the compiler's **COMPILER** vocabulary. The example below adds the flag to both the **COMPILER** and the **INTERPRETER** vocabularies, so that the flag is always available. This extension is usually performed before **CROSS-COMPILE** is executed. The code below generates a flag called **MASTER?** that can be used in any state.

```
ONLY FORTH ALSO C-C DEFINITIONS
0 CONSTANT M/S?             \ 1=master, 0=slave

CC/I                        \ interpreter action
: MASTER?       CC/I  M/S? ;

CC/C                        \ compiler action
: MASTER?                   CC/C  M/S? ;

ONLY FORTH ALSO C-C DEFINITIONS
```

Later on, when the cross compiler is running, condition compilation can be used in a colon definition.

```
: A-WORD
  .....
  MASTER? IF(
    <master-words>
  )ELSE(
    <slave-words>
```

```
)ENDIF
..... ;
```

3.11 Partial Compilation

The partial compilation feature of this compiler reduces the time needed to cross compile applications. It can be seen that every time a cross compilation is carried out certain sections of code, which are never altered, are compiled again and again. This is particularly the case for the supplied files which generate the Forth image. The partial compilation facility allows the user to choose to halt the cross compilation process at strategic positions, to save the symbol table and output image, and then to continue compilation from this position repeatedly.

Two directives are provided. The **SUSPEND** directive is used to halt the present cross compilation and to save the state of the symbol table for later use.The **RESTART** directive will resume compilation after reloading a previously save symbol table file.

SUSPEND — ; <FileName>
"suspend"

Stop the present cross compilation, saving cross compiler information to disc under the given filename, the cross compiler then returning to XShell (or any program that called the compiler). Note that the file name should NOT include an extension, the cross compiler supplies its own. The directive **RESTART** is used to resume cross compilation later.

RESTART — ; <FileName>
"restart"

Continue cross compilation from the position saved under the given file name. The cross compiler saved position files having been generated using the directive **SUSPEND** during an earlier cross compilation.

3.11.1 Example Usage of Partial Compilation

The correct place to use the **SUSPEND** and **RESTART** directives is in the load control file. Here is an example that uses **SUSPEND** to halt compilation and save cross compiler files under the name KERNEL.IMG.

```
CROSS-COMPILE
NO-LOG

ONLY FORTH DEFINITIONS
```

```
DECIMAL  9 15 THRU-USING WRK83      \ system set up
DECIMAL  7    LOAD-USING CODE86     \ main code defs.
DECIMAL 19 76 THRU-USING WRK83      \ main part of Forth
DECIMAL  7    LOAD-USING MSDOS      \ block and disk i/o
DECIMAL 77    LOAD-USING WRK83      \ extensions

SUSPEND KERNEL

FINIS
```

This load control file will suspend cross compilation after the generation of the unexecutable kernel image. At this point cross compilation is effectively frozen and compliler data saved to disc. This allows the cross compiler to be restarted at any time and as often as is required.

Setting up the load control file to restart the cross compilation is best achieved by commenting out the previous load commands and inserting the **RESTART** directive and the new source code to be loaded.

```
CROSS-COMPILE
NO-LOG

ONLY FORTH DEFINITIONS
\DECIMAL  9 15 THRU-USING WRK83      \ system set up
\DECIMAL  7    LOAD-USING CODE86     \ main code defs.
\DECIMAL 19 76 THRU-USING WRK83      \ main part
\DECIMAL  7    LOAD-USING MSDOS      \ block/disk i/o
\DECIMAL 77    LOAD-USING WRK83      \ extensions

\SUSPEND KERNEL

RESTART KERNEL
DECIMAL  2    LOAD-USING ALPHA      \ application files
DECIMAL  2    LOAD-USING BETA       \
DECIMAL  2    LOAD-USING GAMMA      \

FINIS
```

After cross compilation using this load control file is carried out, the final executable image file will be produced. If any of the files ALPHA, BETA or GAMMA contained an error, then of course it is possible to restart the cross compilation again from the suspended position. Once the code is correct, the cross compiler can be suspended after compiling the files ALPHA, BETA and GAMMA, and new source code can be compiled after restarting from this new suspended position. This goes on until the application is completed. The total application development time is decreased, with respect to the time it would have taken without using partial compilation, each time the edit-compile-debug cycle is repeated.

3.11.2 Notes on Partial Compilation

The **SUSPEND** directive flushes and closes the output file, so saving the output image, and then creates three files using extensions .SYM .VAR and .RAM which contain the the symbol table, the cross compiler internal state, and the RAM data respectively. Thus for the selection below:

```
OUTPUT-FILE ROMZ80.IMG
......
SUSPEND KERNEL
```

the saved image is in ROMZ80.IMG and the saved cross compiler data is in KERNEL.SYM KERNEL.VAR and KERNEL.RAM.

Similarly, the **RESTART** directive reloads the cross compiler from the required .SYM .VAR and .RAM files. An output file has to be defined using the **OUTPUT-FILE** directive in the normal way.

This scheme is particularly suitable for use with paged targets. The common kernel can be supplied as .IMG .SYM .VAR and .RAM files, and pages are then compiled without upsetting the original kernel. Note that the .IMG file is not necessary to use **RESTART**.

If partial compilation is being used just to save the time to compile the first part of an application it is often convenient to define the output file and suspended files to have the same name (say KERNEL). After the kernel has been built, the kernel files are then saved.

```
COPY KERNEL.* SAVED.*
```

When running the cross compiler, the saved files should be copied back. This can be done from a batch file, or from within the cross compiler using the **DOS"** word. The use of XCOPY rather that COPY is recommended. The full pathname and extension of the program file are required.

```
DOS" C:\DOS\XCOPY.EXE SAVED.* JOB.*"
......
OUTPUT-FILE JOB.IMG
RESTART JOB
```

When **RESTART** is used, an existing output file will be opened, a new file is only created if one does not already exist.

3.12

Paged Targets

The cross compiler's paged target support is intended to simplify the process of generating applications for targets with extended memory space. Typically these consist of eight bit processors that employ a means of banking/paging memory, such as the Zx80/64180, any 8-bit processor with additional bank-switching hardware, or processors with paged EPROMs such as the 27513. The cross compiler will produce separate image files for each page, during one cross compilation, which are available interactively when blown into separate ROM banks/pages on the target. Note that the paging features of the cross compiler can be used in conjunction with partial compilation.

The most frequently ocurring target paging model is supported by the cross compiler. This model consists of one fixed bank/page of ROM, a variable number of switchable banks/pages of ROM, and one fixed bank/page of RAM.

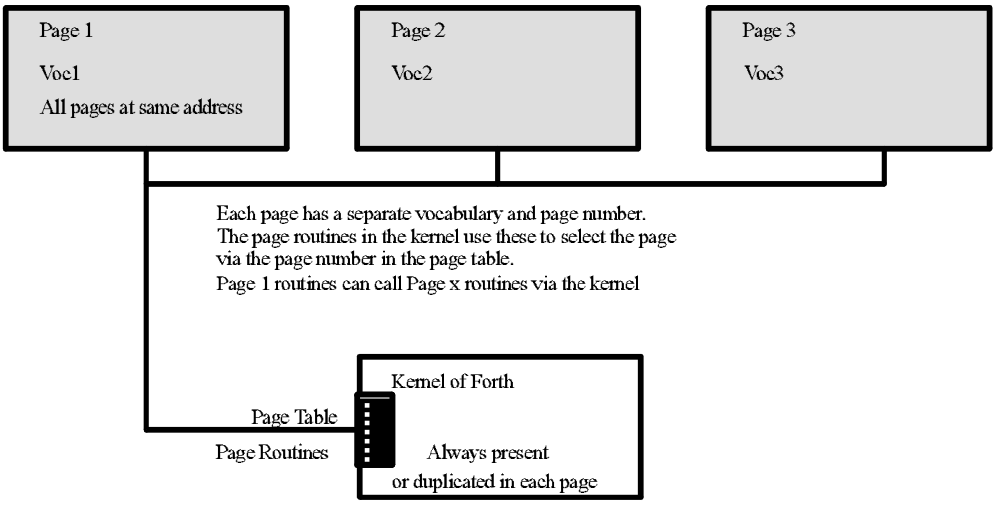
What the paged system effectively does is to give the user access to a Forth system that can be larger than the address space of the processor. The pages themselves are treated as vocabularies, access to the pages being obtained by using the conventional vocabulary word set (For a more detailed description see the RUN TIME SPECIFICATION). As an illustration consider the situation where on a paged target Forth system there resides a page named **VIDEO**. In this page there are a set of words that will position a cursor, display strings, and so on. To gain access to page **VIDEO** the following phrase is used:

ONLY FORTH ALSO VIDEO PAGING DEFINITIONS

Or alternatively just:

VIDEO PAGING

Now by typing **WORDS** all the words in that page will be displayed. Any of the words in this page can now be executed.



Target Paging Model

In the description of paging that follows the fixed ROM shall be referred to as the KERNEL and the ROMs that are banked or paged as PAGES. The cross compiler will automatically compile code suitable the kernel, until directed to compile a page, after which it will return to compiling the kernel.

The paged target specification is divided into two logical phases:

1: During Cross Compilation :- Specifying the logical address at which the page resides, what code is compiled into each page, and the file name under which the page image should be saved.

2: During Target Run Time :- Specifying a means of switching pages, and allowing access to a page's word set.

Both phases have to be completed before a paged target application can be sucessfully generated. The following paged target specification example shows how to accomplish these.

3.12.1 Paged Target Specification

The following examples of source code are available on the cross compiler issue discs. The use of the screen files PAGEA.SCR and PAGEB.SCR, the page constants **PAGEA** and **PAGEB**, and the page vocabularies **PAGEA-VOC** and **PAGEB-VOC** is purely for clarity. The screen files, constants and vocabularies may have any names.

By following the sections on COMPILE TIME and RUN TIME SPECIFICATIONS a paged target application can be generated. Note that for the RUN TIME SPECIFICATION some code may need to written by the user.

3.12.2 Compile Time - the Control file

In the following example a typical load control file to generate a Forth image (in this case for the Zx180/64180) has been modified to include the source files PAGECTL.SCR, PAGING.SCR, PAGEA.SCR and PAGEB.SCR. The PAGEA and PAGEB screen files are just this example application's name for the files that contain source code for its pages. The files PAGECTL and PAGING must be included within any paged application generation

CROSS-COMPILE

```
ONLY FORTH DEFINITIONS
DECIMAL 9 15 THRU-USING WRK83
DECIMAL 1 LOAD-USING CODE180
DECIMAL 1 LOAD-USING ROM-IO
DECIMAL 19 67 THRU-USING WRK83
DECIMAL 1 THRU-USING ROMTOOLS
DECIMAL 2 LOAD-USING PAGECTL        \ PAGING FILE
DECIMAL 2 LOAD-USING PAGING         \ PAGING FILE
```

```
DECIMAL 2 LOAD-USING PAGEA
DECIMAL 2 LOAD-USING PAGEB

FINIS
```

This example load control file will generate a paged target system.

3.12.3 The PAGECTL File

The following example of a PAGECTL file will be similar across different targets. The only differences are in the values of the page constants for pages **PAGEA** and **PAGEB**, these are the values needed by the target's memory management unit (MMU) to bank/page pages, and should be set by the user.

```
HEX

000 CONSTANT PAGEA
0FB CONSTANT PAGEB

VOCABULARY PAGEA-VOC
VOCABULARY PAGEB-VOC

CREATE PAGE-TABLE
    PAGEA C,
    PAGEB C,
END-PAGE-TABLE

DECIMAL
```

The above example paged target specification sets up two pages. Pages are defined by declaring a page name constant which returns the page number passed to the MMU, and declaring a vocabulary that will contain a page's words. The pages are then set in a table named **PAGE-TABLE**, which should contain all the target's pages. The maximum number of pages is 255.

3.12.4 The PAGEA and PAGEB Load Files

The cross compiler has to be directed to cross compile a separate page. Following is an example load screen that is used to do this. Two directives are needed:

COMPILE-PAGE Page-Addr — ; <FileName>
"compile-page"

This word will cross compile a page that is executable from address PAGE-ADDR, the resulting image file then stored to disc as <FILENAME>.

FINIS-PAGE —
"finis-page"

This word is used to finish off the compilation of a page, and return to cross compiling the kernel.

```
HEX
08000 COMPILE-PAGE PAGEA.IMG

ALSO PAGEA-VOC DEFINITIONS

3 LOAD

FINIS-PAGE

PREVIOUS DEFINITIONS
DECIMAL
```

Notice that before the source code is cross compiled with the phrase:

```
3 LOAD
```

the page's vocabulary has to be set as the definition vocabulary with the phrase:

```
ALSO PAGEA-VOC DEFINITIONS
```

This is essential and allows the page to be available interactively to the target.

3.12.5 Run Time - the PAGING file

This file contains the target primitives that control page switching. These words are **PAGE! PAGE@ PAGING** and **(INIT-PAGES)**. These words have to be present, although the code for some of them may have to be written by the user. The code for **PAGE@** and **PAGE!** has to be written by the user. These words return and set the the page register or MMU. **PAGING** is a high level word used during vocabulary switches and itself uses **PAGE!**. The default action of the deferred word **INIT-PAGES** is **(INIT-PAGES)**, which has to initialise the page register or MMU.

It necessary to be able to read the page register or MMU. If the hardware does not permit this, **PAGE!** should write a copy into a RAM location for later use by **PAGE@**.

3.12.6 Paging Glossary

Note that only these words need to be provided on the target.

(INIT-PAGES) —
"paren-init-pages"

This word has the default action of doing nothing. Some MMUs may require initialisation, and if this is the case then the user will have to write the code needed to do this.

PAGE! physical-page# —
"page-store"

This word will select physical-page# on the MMU. The actual code for this word will have to be written by the user if none is provided for a particular MMU.

PAGE@ — physical-page#
"page-fetch"

This word will read the current page from the MMU. The actual code for this word will have to be written by the user if none is provided for a particular MMU.

PAGING —
"paging"

Used when setting a vocabulary search order and definitions vocabulary. See the words **ALSO**, **ONLY**, and **DEFINITIONS** in the target glossary. This extra word is needed when the words for a particular vocabulary are in a page/bank. In the following example the words for the vocabulary **PAGEA** are in another page and so **PAGING** is needed to select that page:

ONLY FORTH ALSO PAGEA PAGING DEFINITIONS

3.12.7 Further Notes on Paged Targets

The only words that can be executed at any given time are those which are in the kernel or the current page. To execute a word in another page from the kernel requires the page to be paged in (see section on RUN TIME SPECIFICATION) and then executed. Executing words in one page from another page has to be done indirectly via the kernel. A example is given in the PAGING.SCR file.

For further paged target information see the chapter on EXTENDING THE CROSS COMPILER.

3.13

EPROM boundary control

Occasionally it is necessary to provide a target on non-contiguous EPROMS, as part of the memory map is used for special purposes. This may occur because memory is only partially decoded, or because a monitor is present. Such situations are often found in the domestic and leisure computers for which the facility was originally designed. More recently, the use of paged EPROMs has meant that boundary control can be important.

Under these conditions it is also possible that the first or last few bytes of the EPROM may be reserved for special use, and must be skipped by the compiler. We have only needed this facility when generating applications for domestic computers.

To cope with these requirements the word **BOUNDING** is used. It needs four parameters an is used with the syntax:

n1 n2 n3 n4 BOUNDING

The first parameter n1 declares the size of the EPROM in bytes.

Parameter n2 is a bit mask identifying which address lines select the EPROM. For example an 8k EPROM of size 02000 (hex) is controlled by address lines A15,14,13 and will have a mask value of 0E000 (hex).

Parameter n3 specifies how far before the end of the EPROM the compiler stops to ask the user whether it should step to the next EPROM. This limit is used to prevent the last word in an EPROM being compiled across an EPROM boundary. If the answer is no, the qustion will be repeated before each word is compiled until the answer is yes or the EPROM boundary is crossed. When the yes answer is given, the compiler steps to the start of the next EPROM, and repeats the question. If the answer is again yes, the complete EPROM is skipped. Note that specifying a value of say 0100 does guarantee to reserve 0100 free bytes at the end of the EPROM, it only specifies at what point the compiler starts asking what needs to be done.

Parameter n4 specifies how many bytes are skipped at the start of each EPROM.

For example, using 8k 2764 EPROMs, starting checking 0100 bytes before the end of each EPROM, and skipping no bytes at the start of each EPROM requires the following declaration:

```
HEX
02000 0E000 0100 0 BOUNDING
```

For common steps the following values of size and mask apply.

size #bytes	**bit-mask**	**common EPROM**
32k	08000	08000 27256
16k	04000	0C000 27128
8k	02000	0E000 2764

4k	01000	0F000	2732
2k	00800	0F800	2716

3.14 Controlling the Symbol Log

The symbol table log is a report of all words, labels constants etc. generated by the cross compiler. This log can be turned on or off with the directives LOG and NO-LOG respectively. The default cross compiler setting is for a full symbol table log displayed on screen. The symbol table log can be redirected to file or printer with the directives **FILE:** and **PRN:**, and this can be done from the DOS command line as for example:

X64180 ROMCTL FILE: LOGGED.TXT 2 LOAD

When entered from the DOS command line this will invoke the cross compiler and direct the symbol table log to a file called LOGGED.TXT. To redirect to the printer the command line would be:

X64180 ROMCTL PRN: 2 LOAD

3.14.1 Symbol Log Glossary

The symbol table log directives are:

LOG —
"log"

Generate a full symbol table log.

NO-LOG —
"no-log"

Generate a reduced symbol table log.

CON: —
"to-con"

Direct the symbol table log output to the screen. The default output device. If required this directive has to be used before the command **CROSS-COMPILE**.

FILE: — ; <filename>
"to-file"

Direct the symbol table log output to the file FileName. If required this directive has to be used before the command **CROSS-COMPILE**.

PRN: —
"to-prin"

Direct the symbol table log output to the printer. If required this directive has to be used before the command **CROSS-COMPILE**.

3.15 Command Line Interpretation

Running the cross compiler can be fully automated through the use of batch files. For example, entering the following command line at the DOS prompt will invoke the cross compiler, which will then use the load control file ROMCTL for compilation, before returning to the DOS prompt.

```
X64180 ROMCTL  2 LOAD
```

This command can be inserted into a batch file, or be run from the XShell program (the cross-compiler command string for XShell should be set to this command). On the cross compiler issue discs there is a batch file called XC.BAT which contains a similar command suitable for the supplied target.

See the SYMBOL LOG section of CONTROLLING THE CROSS COMPILER for other directives that can be used from the command line.

The method used to interpret the command line is simple. The command tail (the remaining text of the command line not including the program name) is treated as a line of Forth code and interpreted as such with one exception. This is that the first word is treated as a file name and an attempt is made to open a file of this name. If this attempt fails then the word is interpreted, along with the rest of the line as Forth code. If the attempts succeeds then the file becomes the current file, and only the rest of the line is interpreted. This means that the user can enter any Forth code on the command line, although in practice only the directives **LOAD** and **THRU** to interpret source code, and the **FILE:** and **PRN:** directives (see the SYMBOL LOG section of CONTROLLING THE CROSS COMPILER) are used.

Communicating with the Target

The developer has to communicate with the target in two forms. Firstly, the target image must be transferred to the target processor so that it can be executed and tested. Secondly, a means of interacting with the target Forth must be established, usually by means of a serial link.

4.1 Transferring the Image

There are four standard methods of transferring the memory image to the target. In terms of turn-round time, a parallel connected EPROM emulator is heartily recommended.

1. EPROM Programmer
2. EPROM Emulator
3. Processor Emulator
4. Downloading to a monitor

Since Forth is an interactive language, the time taken to turn around an iteration of the cross compiler is important. Time spent compiling and transferring code is time spent NOT debugging the application. For reasons such as these the method of working will depend on the speed of the tools at hand. Do not be afraid to modify to the working patterns to find the most satisfactory routes.

4.1.1 EPROM Programmers

Transferring the image by means of EPROMs is perhaps the commonest way of working. The binary image file produced by the compiler is in a form suitable for many PC plug-in programmers, such as the commonly available ‘Sunshine’ series. These programmers, especially when used with the Intel ‘QuickPulse’ algorithm, provide an economical form of transfer.

If a serially connected programmer is used, the XC-COMM tools include Intel Hex and Motorola S-record downloaders. In this case ensure that these tools are compiled onto the PC PowerForth as a utility, which can then be installed into XShell2 on one of the two user defined functions.

Note that if the serial line transfer or EPROM programming algorithm is slow, the time taken to transfer the image and program the EPROM can considerably exceed the time taken to cross compile the image.

4.1.2 EPROM Emulators

A PC plug-in EPROM emulator provides the fastest possible means of transferring code to the target. These devices transfer code at about 10k bytes/second, and have many advantages over EPROM programmers. The Leburg LePROM Emulator is directly supported by the cross compiler, which can be directed to compile directly into it, so avoiding even the transfer time. See the chapter on 'Controlling the compiler' for details of the directives.

The use of EPROM emulators during development has proven to be a great time-saver. Additional benefits include reduced wear and tear on IC sockets and no broken IC legs.

4.1.3 Processor Emulator

Processor emulators are valuable at the very early stages of bringing up a Forth system, or for debugging complex assembler routines, but the interactive nature of Forth reduces their value once Forth itself is running. This factor, allied with the expense of most emulators, makes them most suitable as departmental items, rather than items for use with every PC.

Most emulators are connected serially, and download to them is usually performed using the XC-COMM tools. Since these allow the choice of the PC's COM1 and COM2 ports, the target's serial line can be connected to one port, and the emulator's to another. The **Terminal Mode** terminal emulator in XShell v2 can then be used to control both the emulator and the target. When using this configuration it is helpful to write additional words in the XShell environment that allow rapid switching. The same comments apply on the subject of serial transfer speed as were made about EPROM programmers.

4.1.4 Downloading to a Monitor

This situation is convenient when installing Forth on a commercial Single Board Computer (SBC), as monitor services can often be used by the application. Communication with these monitor usually requires the XC-COMM

tools as well as **Terminal mode**. The comments about the speed of serial download apply again.

4.2 Interacting with the Target

Once the target Forth is running, a link between it and the PC is required. The tools to be used will vary according to the style of source code (text files or screen files) preferred. The very lowest level required of the comms tool is dumb terminal emulation. The Terminal Mode (**<ALT> T** or **<ALT> S**) built into XShell2 v2 is quite adequate for this purpose.

4.2.1 Comms Requirements

What sort of communications package is needed is often determined by the mechanism used to transfer the memory image to the target. If very rapid image transfer is available, and the target processor is not very powerful, it is much quicker to re-run the cross compiler and reload the image, than to download a significant quantity of source code down a serial line. However, if the EPROM programmer or transfer method is slow, and only small quantities of source code are to be tested on each iteration, downloading source code to the target is a good option.

For many users, the design and evolution of the working method is thus determined by several factors. The XShell2 and T-COMM tools provide several ways of dealing with each component.

4.2.2 Working with Text files

If standard text files are being used, any communications program can be used in addition to XShell v2. Suitable ones are Crosstalk, Mirror, and Procomm. They may also be used to download code to the target systems.

If source code is to be downloaded the following changes to the target code for **EXPECT** can be recommended if they are not already installed on your target (they may be found in T-COMM.SCR).

1. Add a flag **ECHO-FLG** that controls echoing of the received data.

```
......
ECHO-FLG @
IF  EMIT  ELSE  DROP  ENDIF
......
```

2. Use this same flag to add flow control add the start and end of **EXPECT**.

```
......
XON EMIT
<main EXPECT loop>
XOFF EMIT
......
```

3. Write words to enable and disable echoing.

```
: ECHO
  ECHO-FLG ON  ;

: NO-ECHO
  ECHO-FLG OFF  ;
```

4. Add these words to the start and end of files to be downloaded. Disabling any echoing reduces the work load of both the target and the host, so increasing compilation speed.

5. Tune the comms package. The XON/XOFF flow control should be faster than having to add delays to the end of each line. Not all target processors are capable of accepting streams of characters at 4800 baud or above using un-buffered input systems. In such a case either accept a slower baud rate, or write an interrupt driven and buffered input routine. The buffer need be be no longer than the maximim line length - 128 bytes will be plenty.

4.2.3 Working with Screen files

The **Terminal Mode** of XShell v2 has support for handling screen files using the host as a disc server. The additional target code may be found in T-COMM.SCR. The functions are detailed in the XShell v2 manual and the screen file itself. Once these extensions are loaded on the target, the words

LOAD and **THRU** may be executed on the target, and will load screens down the serial line. If the target processor's UART will support it, running the serial line at 38400 baud will enable transfers at up to 4 screens per second.

Blank Page

Debugging the target

5.1 Introduction

This section covers debugging a Forth system from scratch. We are consequently covering ground sometimes accompanied by the sounds of gnashing teeth and language unsuitable for delicate ears. In practice however, starting a Forth system from scratch is straightforward, provided that the hardware is already working. Despite this proviso, we have used these cross compilers to produce the first software for a range of processor cards, and would now prefer to 'bring up' a new processor card using Forth rather than assembler. The reason for this is that Forth code definitions are short. They are thus more likely to work first time.

Debugging a target system that does not even sign on can require an understanding of how Forth works. Despite it's reference to a different dialect, the best description of the internal workings of Forth is still the 'Systems Guide to Fig-Forth' by Dr C.H. Ting (Offete Enterprises). Also useful may be 'Threaded Interpreted Languages' by Loeliger (Byte/McGraw Hill).

The PowerForth target code is written so that very few machine instructions are executed before Forth is started. These instructions set up the stacks, and the user and interpretive pointers, and then execute the inner interpreter routine **NEXT**. The start up machine code sequence can be debugged by inserting a subroutine call to a routine that displays a character.

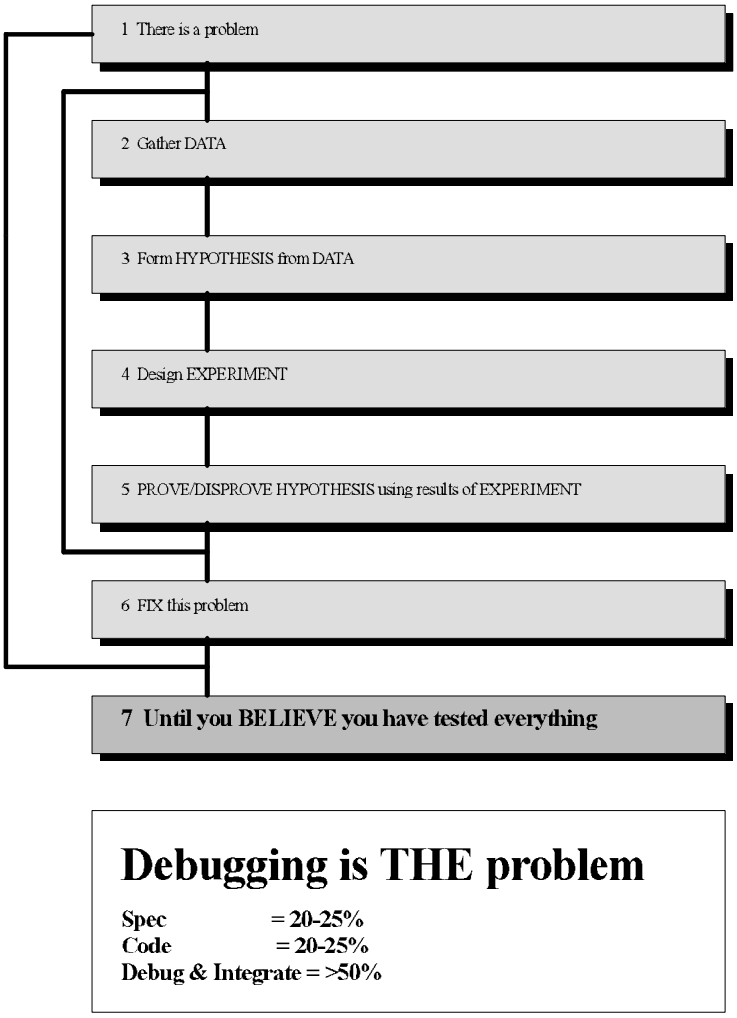
5.2 Debugging strategy

The best strategy in debugging is the same one advised by both Julius Caesar and Clausewitz - DIVIDE and CONQUER. It is very much easier to track down errors in small pieces of code. Debugging is usually considered to be a minor art form whereas it actually requires rigorous application of traditional scientific method. Considering that software audits imply that debugging and system integration is the largest part of any software project, there is remarkably little literature on the subject.

First: gather evidence. What symptoms can you find. Examine them.

Second: form a hypothesis. Provide a possible explanation for these symptoms. A hypothesis predicts a result that you have not seen before (usually that the bug goes away).

Third: design an experiment to prove this hypothesis. The experiment should give a yes/no answer. A 'definite maybe' is not good enough. This is actually



The Formal Debugging Strategy

the hardest part. Since there may be (will be) several interacting faults in a complex piece of code, isolating them one a time from the 'noise' of other faults can be difficult.

Fourth: Prove the hypothesis using the results of the experiment. In many instances the experiment is just a paper exercise of systematic analysis of previous results.

Fifth: Fix it! Do not fix it until you have hard evidence from your experiment. In some instances the experiment may serve to isolate or eliminate candidates, so that more than one experiment may be necessary to isolate the problem.

Debugging also requires a certain attitude of mind. The cry 'but it MUST work' usually indicates that the investigator is looking in the wrong place, or is ignoring information because of an initial assumption. This cry is often produced when code is being debugged by the person who designed or coded it. Debugging is very much faster with two people, even if the second person is only there to listen to the explanations of the other person as to why the bug can't possibly exist in the first place because ... oh whoops!

5.3 Start Up and Initialisation

There are two sections of initialisation code that must be considered when bringing up Forth for the first time on new hardware. First there is the initialisation of the registers and/or memory that correspond to the Forth virtual machine, then there is the initialisation of memory and data structures used by Forth. Alongside this is the initialisation of hardware devices, of which the most important at this stage is the UART used for communications with the PC host.

5.3.1 Initialisation of Forth itself

Once Forth is running the order of execution of Forth words is usually **STARTUP COLD ABORT QUIT**. Most of the initialisation is performed in **STARTUP** (often by **(INIT)**), with the first sign-on message appearing in **COLD**. If your system does not sign on at all, the most likely faults are that **EMIT** and **TYPE** do not not work, or that Forth itself has not been started. If Forth itself has not been started, it is likely that reset vectors are not correctly set. ROM based PowerForth targets will often perform all their initialisation within **(INIT)** and **COLD**, and **STARTUP** may not exist. The word **(INIT)** is mostly involved with initialising the interactive parts of Forth, and so may not be present in the minimal targets designed for use with Umbilical Forth.

5.3.2 Initialising the UART

Once a routine to transmit a character has been written, the rest of a Forth system can be debugged. Four simple routines have to be written.

1) AINIT - Initialise the UART and Baud Rate Generator
2) AEMIT - Transmit a character - used by **EMIT**
3) AKEY - Read a character - used by **KEY**
4) AKEYQ - Test for a received character - used by **KEY?**

If these routines are written in code, they can be used even before the Forth machine is working. For example, in a 6801 implementation the routine AEMIT may be written as an assembly language subroutine. The following woords may then follow.

```
ASSEMBLER
L: AEMIT

  ..... RTS
FORTH

CODE (EMIT)      \ char —
PULD   BSR AEMIT
JMP NEXT   END-CODE

EMIT             \ char —
(EMIT)   1 OUT +!  ;

TYPE             \ addr len —
BOUNDS
?DO  I C@ EMIT  LOOP  ;
```

The point of this example is that the assembly language routine **AEMIT** can be used both at high and low level. Since it does very little it will be short, and more likely to work first time. It is a great pity that the authors of data books have lost the habit of writing code examples.

5.3.3 Debugging the Forth

By writing the very simplest possible versions of **EMIT** and **TYPE** characters can be displayed as the code executes. Until Forth is running with **(EMIT)** working properly, you are still in the world of debuggers, emulators, soldering irons, and the usual knife and fork work. Once **(EMIT)** is working, the usual practice is to insert phrases such as

```
ASCII A (EMIT)
```

in the source code to indicate progress through the start up sequence. Note that for this to work properly not only is **EMIT** working but also **LIT**, the run-time action of a literal, and also the Forth chaining system. A liberal sprinkling of such phrases helps to isolate the fault area very quickly. Then **EMIT** itself can

be tested. Once **EMIT** is working it is likely that the user variable mechanism is also working.

5.4 Common Sources of Error

1. Incorrect initialisation either of registers, or of hardware devices, interrupts, UARTs and so on. The solution to this problem is to read the manual again, or to find some working code from another application.

2. Incorrect use of a target operating system or monitor, especially with respect to its register or stack use. A sympton of this may be that the first character only is issued.

3. Forgetting to change the number base correctly. It is good practice to prefix hexadecimal numbers with a '0' so that

020 and 0100

are hexadecimal numbers, whereas

20 and 100

are decimal numbers. In this way a missing or incorrect base declaration becomes more visible in the source code. It is also good practice to include a base declaration (**HEX DECIMAL** or **BINARY**) in any screen in which it matters. Also remember that placing one of these words inside a colon definition has NO effect during compilation, it will only execute on the target. Such problems are particularly likely in team projects.

4. If you have changed the stack organisation, the limit checks in **?STACK** may no longer be appropriate. The usual symptom of this is a stream of error messages followed by a total crash. This problem often occurs when the ROM/RAM partitioning is non-standard.

Blank Page

PowerForth Internals

This section of the manual describes some of the internal workings and structures of the standard PowerForth implementation.

6.1 User area layout

The user area of a task is an area of memory usually reserved for variables that may differ from task to task in a multi-tasking environment.

For instance, two tasks may be preparing numbers for output. This will require them to use a buffer area known as **PAD**. Each task should have its own area to avoid corruption by other tasks. Variables such as **BASE** should also be independent. It is for this reason that separate user areas can be created for each task.

When a user variable is defined:

n USER <name>

the value of n is stored. When ame is later executed, the address returned by **<name>** is calculated by adding n to the base address of the user area, which is held either in memory, or in a processor register. The base address may be different for each task, or several tasks can share one user area.

The user area is defined by a screen in the PowerForth WRK83.SCR source code file.

The user area is usually 128 (80h) or 256 (0100h) bytes long, so allowing variables in the range 0..0FEh to be used. The six byte area at the beginning is reserved - don't use it unless desparate for RAM space. A summary of the use of the user variables follows. The first part of the user area is necessary to allow numeric output in different bases, which requires the user variables including **BASE** and **HLD** as far as (but not including) **PAD**. String handlers use **PAD** so that the **PAD** areas must be kept separate is scheduling can occur while **PAD** is in use. Similarly the TIB areas must be kept separate.

S0 contains the address of the top of the data stack. Stacks grow downwards in memory all but a few implementations. This address is used to reset the stack pointer during initialisation or error recovery.

R0 contains the address of the top of the return stack. Stacks grow downwards in memory in all but a few implementations. This address is used to reset the stack pointer during initialisation or error recovery.

'TIB contains the base address of the terminal input buffer.

WIDTH contains the maximum name field width. This word may well disappear in later versions of PowerForth.

FENCE contains an address below which words cannot be forgotten. This will become a normal variable in later versions.

DP contains the address of the end of the dictionary. This will become a normal variable in later versions.

VOC-LINK contains a pointer into the most recently defined vocabulary. This will become a normal variable in later versions.

BLK contains the block number last loaded, where 0 means the keyboard.

IN contains the current offset from the start of the terminal input buffer.

OUT contains the current output character position in the line.

SCR contains the screen number of the screen last **LIST**ed.

SPAN contains the number of characters last read into **TIB** by **EXPECT**.

HLD is used during number conversion to point at the last character generated.

CURRENT points to the vocabulary into which words are defined. This will become a normal variable in later versions.

STATE is non-zero during compilation. This will become a normal variable in later versions.

BASE contains the number conversion base.

DPL contains the number of digits after the decimal point or double number marker, -1 meaning no marker.

#TIB contains the number of characters in the terminal input buffer.

CSP is used by the compiler during error checking. This will become a normal variable in later versions.

CONTEXT is an array of up to eight vocabularies which are searched in order. See the tutorial section on vocabularies. This will become a normal variable in later versions.

#L contains the number of cells last returned by **NUMBER?**.

TAB-WIDTH contains the tab separation in characters.

#D contains the number of digits last converted by **NUMBER?**.

LINE# contains the line number on the page. It is incremented by **CR** and cleared by **PAGE**.

PAGE# contains the output page number. It is incremented by **PAGE** and must be cleared by the user.

PAD is a buffer used for string handling. Numeric text conversion is performed immediately below **PAD**.

6.1.1 User Variables for Small Processors

The implementation for some processors separates the conventional set of USER variables into two sets. The first set is the set of variables used for compilation and text interpretation, and this set is implemented as conventional variables. The second set consists of all the others, and these are implemented as conventional user variables.

The reason for this change is that for a single user system (only one user compiling code), it is only a waste of RAM space to duplicate these variables for each active task.

6.2 Vocabularies

The function and use of vocabularies in a Forth program is more fully discussed in the PC PowerForth manual. This section explains how vocabularies are implemented in the PowerForth targets.

The vocabulary mechanism used in PowerForth is a derivative of that proposed by Bill Ragsdale in the Forth-83 Standard. The mechanism is compatible with previous systems, while permitting control of the search order. With the passage of time this mechanism for search order control has been generally accepted.

The address of the vocabulary into which words are compiled is held in the user variable **CURRENT** and an address of the first vocabulary to be searched for word names is held in the user variable **CONTEXT**.

CONTEXT is actually an array of up to eight vocabulary addresses, which are searched in order. A zero entry indicates no vocabulary. The word **FIND** simply scans along the array, searching each vocabulary in turn. The order in which vocabularies are searched is thus defined by the order of the vocabulary entries in the **CONTEXT** array.

A very small vocabulary called **ROOT** contains a few definitions that allow the user to switch vocabularies. This is the last entry and is (almost) never deleted from the search order.

The data held by each vocabulary consists of several fields, the most important of which are a field containing the number of threads in the vocabulary, and the field containing the name field address of the last entry in each thread. When the dictionary is searched, each vocabulary in the search order is inspected in turn. A hashing algorithm determines which thread will be searched. This same algorithm is used by **$CREATE** to define the thread in which the word is placed. The cross compiler defaults to creating 16 threads for each vocabulary.

6.3 PowerForth dictionary structure

PowerForth (cross compiler versions 4.0 and later) uses a multiple thread vocabulary mechanism that gives extremely high compilation speeds. All Forth words use the same dictionary entry structure, which consists of four fields.

1. Thread link field (two bytes/one cell)
This field contains a pointer to the previous word in this search thread. When a word is searched for, the word name is hashed to select one of n threads (may be different in each vocabulary), and all words are linked by one of these threads. A pointer value of zero indicates that there is no previous entry.

2. Name field (variable length)
The name field consists of a count byte, followed by the ASCII text of the word name. The bottom five bits of the count byte contain the actual count, and the top three bits are used as follows:

Bit 7: Always set
Bit 6: Set if word is IMMEDIATE
Bit 5: Set if word is hidden from search

3. Code field (normally three bytes)
The code field contains executable code. This code may be one of a number of short routines, and defines the type of word that is described by the contents of the parameter field. In a **CODE** definition there is no distinction between code field and parameter field as executable code extends as far as is necessary. For most other definitions the code field contains a jump or a call to the routine that handles the run-time action of the defining word that created the definition.

4. Parameter field (variable length)
This field contains the remainder of the definition of the word. For a **CODE** definition this will be executable code as a continuation of the code field. For a

normal colon definition this will be a list of other Forth words. For **VARIABLE**s and **CONSTANT**s this will be the value of the variable or constant.

6.4 Threading structure

PowerForth uses Direct Threaded Code (DTC) on all processors with conventional architectures. The exceptions are for the Harris RTX-2000 and Novix NC4016 whose hardware manages threading as subroutine calls, and for the 1802-6 series.

In a direct threaded system the code field of a definition always contains executable code. For the majority of words this is a jump or a call to the routine that handles the run-time action of the defining word used to create the definition. However, for **CODE** definitions this routine is the body (parameter field) of the definition and so no jump is required. DTC is an efficient trade off between speed and memory space for the majority of small processors, and runs considerably faster than Indirect Threaded Code (ITC). For details of the exact usage see the assembler chapter of the target manual.

Blank Page

Umbilical Forth

7.1 What is an Umbilical Forth?

Umbilical Forth is software that makes it practical to install Forth on a processor with very limited memory resources. Such processors are often used in single chip applications. Umbilical Forth is only supplied with targets used for this type of role.

Forth is an easily ported language, and because of that portability, Forth is provided on many different microprocessors. However, some processors do not lend themselves to the Forth ideal of interactive development directly on the target. This can often happen in the case of processors with small on-chip EPROM spaces such as single chip controllers like the 68HC11 or TMS7000 series, or those with separated Program and Data addressing (Harvard architecture), such as the Hitachi HMCS400 and the Intel 8051 families..

Umbilical Forth grew out of the need to provide an interactive Forth on a Hitachi H408 processor (a 4 bit processor with separate addressing of 10 bit wide program and a 4 bit wide data space) which could only be accessed via an emulator. This processor exemplifies the limitiations discussed above.

The theory of Umbilical Forth is to provide an interactive Forth language by the integration of a Forth on a host and a Forth on the target. The Forth on the target will run the bare minimum of code sufficient to run the application and a message handler to the host, while the host Forth will act as both interpreter and compiler, as well as providing disk services. The host Forth is an extension of the MPE Cross Compiler (version 4.0 or above). It simulates the interactive portion of a target Forth by searching for words in the cross compiler's symbol table, and reading and writing to the target memory. Both of these Forths integrate to provide the usable and seamless single Forth. The name Umbilical Forth arises from the link required between the host and the target, typically provided by a serial communications cable.

An MPE Special Target version of Umbilical Forth contains a fully preinstalled cross compiler in the COMPILER directory, and an example target system in the CHIP directory. The CHIP directory also contains an XShell2 configuration file that will automatically call the compiler for you using the **F3** key.

7.2 Distribution Files

Umbilical Forth is supplied preinstalled for a target system. There are source files which provide the extensions to both the cross-compiler and target code. The files issued over and above the normal cross compiler and target files are described below. Unless you wish to customise the compiler itself, you will not have to use these files in normal operation of the cross compiler.

7.2.1 Target files

The source files needed to generate the target code (over and above the normal target files) are:

TARG-END.SCR	The target end of the Umbilical link software.
MESSAGES.SCR	This file defines the message code values used by the link.

The target Forth itself is compiled from the files in the usual way from a set of files called:

ROMCTL.SCR	Configuration, memory control, and load control file which includes the files described above. This file contains all the relevant target code set up directives for the cross compiler, and finally starts the interactive Umbilical Forth link.
MINCODE	Code definitions for the processor.
KERNEL.SCR	High level definitions. This file will be missing the usual interpreter/compiler layer provided with conventionally interactive targets.
ROM-IO.SCR	Serial I/O drivers. These drivers are interrupt driven, and will put the processor into an ‘IDLE’ or ‘SLEEP’ mode while waiting for a character.

These two sets of files compile a reduced Forth word set from which the conventional interpreter and compiler layers have been removed. The file TARG-END.SCR is the target side of the message passing and message execution code.

7.2.2 Host files

It is most unlikely that you will ever have to use the source code of the compiler itself.

The issued source files assume that the host is an MPE cross compiler (version 4.0 or later) built in the usual way. After installation, a separate directory called UMBxxx or SOURCE will be found in the COMPILER directory. The source files needed for the host are:

UMB-CON.SCR	The file UMB-CON is the load control file for the host side of Umbilical Forth and is all that need be loaded.
PC-COMMS	The PC-COMMS file contains code for using the serial port on IBM PC's or compatibles.
MESSAGES.SCR	This file defines the message code values used by the link.
HOST-END.SCR	The HOST-END file is the host side of the message passing/execution system.
EMULATOR.SCR	The EMULATOR screen file contains the source code for the interpreter/compiler part of Umbilical Forth.

7.3 Installation on the target

The target end of Umbilical Forth is generated by adding extra files that need to be loaded to the ROMCTL file, for example:

```
DECIMAL 2 LOAD-USING MESSAGES
DECIMAL 2 LOAD-USING TARG-END
```

The target end of Umbilical Forth will now have been loaded. See the ROMCTL.SCR file for the real thing. You will find a preinstalled version in the CHIP directory.

7.4 How to use it

The aim of Umbilical Forth is to give the illusion to the user that the Forth being used is a single entity. In fact there are two Forths running, one on the host and one on the target, which combine to give the desired effect.

To use Umbilical Forth all that has to be done is to enter **n BAUD UMBILICAL-FORTH** on the host. This is usually done from the ROMCTL.SCR file at the end of cross compilation. Umbilical Forth will now run and return the prompt “T:OK” from now on, until the word **BACK** is entered and the normal Forth prompt “ok” appears. **BYE** can be used to leave the system in the normal way.

The following word set is a reduced version of the normal PowerForth interactive word set. It is a sufficient base for most applications, and reduces the target Forth kernel to less than 2k bytes. If you need the extra space, the code definitions for your target processor can be rewritten to optimise them for space rather than speed. One of our more desparate customers claims to have reduced a Forth kernel word set to 512 bytes on an RCA 1802.

Many words provided by a normal interactive Forth interpreter are simulated by special cases on the PC side. A complete list can be obtained by typing **WORDS** which shows two groups. The first is the set of words which have been defined during cross compilation. This set includes the run time actions of defining words whose defining action is performed on the PC side. The second set includes all the words for which special cases occur during interpretation.

7.4.1 Reduced Forth word set

Note how the compiler layer is simulated by the host side, and does not have to be included into the target Forth kernel.

The fully commented action of these words is contained within the cross compiler and target manuals.

! n addr —

(+LOOP) n —

(;CODE)

(?DO) n1 n2 —

(DO) n1 n2 —

(INIT) — ; main initialisation

(LOOP) —

(OF) n1 n2 — [equal] | n1 n2 — n1 [not equal]

***** n1 n2 — n3

+ n1 n2 — n3

+LOOP —

Available via the Umbilical Forth target emulator (on host).

***-** n1 n2 — n3

/ n1 n2 — n3

/MOD n1 n2 — rem quot

0 — 0

0> n — t/f

0= n — t/f

0< n — t/f

1+ n — n+1

1- n — n-1

: —
Available via the Umbilical Forth target emulator (on host).

;S —
Replaced by **EXIT**

< n1 n2 — t/f

> n1 n2 — t/f

= n1 n2 — t/f

>R n —

?BRANCH —
Available via the Umbilical Forth target emulator (on host).

AND n1 n2 — n3

BEGIN —
Available via the Umbilical Forth target emulator (on host).

BRANCH —

C! b addr —

C@ addr — b

CASE —
Available via the Umbilical Forth target emulator (on host).

CMOVE addr1 addr2 len —

CONSTANT n —
Available via the Umbilical Forth target emulator (on host).

DO n1 n2 —
Available via the Umbilical Forth target emulator (on host).

DOES> —
Available via the Umbilical Forth target emulator (on host).

DROP n —

DUP n — n n

ELSE —
Available via the Umbilical Forth target emulator (on host).

ENDCASE —
Available via the Umbilical Forth target emulator (on host).

ENDIF —
Available via the Umbilical Forth target emulator (on host).

ENDOF —
Available via the Umbilical Forth target emulator (on host).

EXECUTE addr —

I — n

IF —
Available via the Umbilical Forth target emulator (on host).

LIST n —
Available via the Umbilical Forth target emulator (on host).

LIT — n

LOAD n —
Available via the Umbilical Forth target emulator (on host).

LOOP —
Available via the Umbilical Forth target emulator (on host).

M/MOD d n — rem quot

NOT n1 n2 — n3

OF —
Available via the Umbilical Forth target emulator (on host).

OR n1 n2 — n3

OVER n1 n2 — n1 n2 n1

R0 — addr

R> — n

R@ — n

REPEAT —
Available via the Umbilical Forth target emulator (on host).

RESET-STACKS —

ROT n1 n2 n3 — n2 n3 n1

RP! n —

RP@ — n

S0 — addr

SP! n —

SP@ — n

SWAP n1 n2 — n2 n1

U< u1 u2 — t/f

U> u1 u2 — t/f

UM* u1 u2 — ud

UM/MOD u31 u16 — urem uquot

UNTIL —
Available via the Umbilical Forth target emulator (on host).

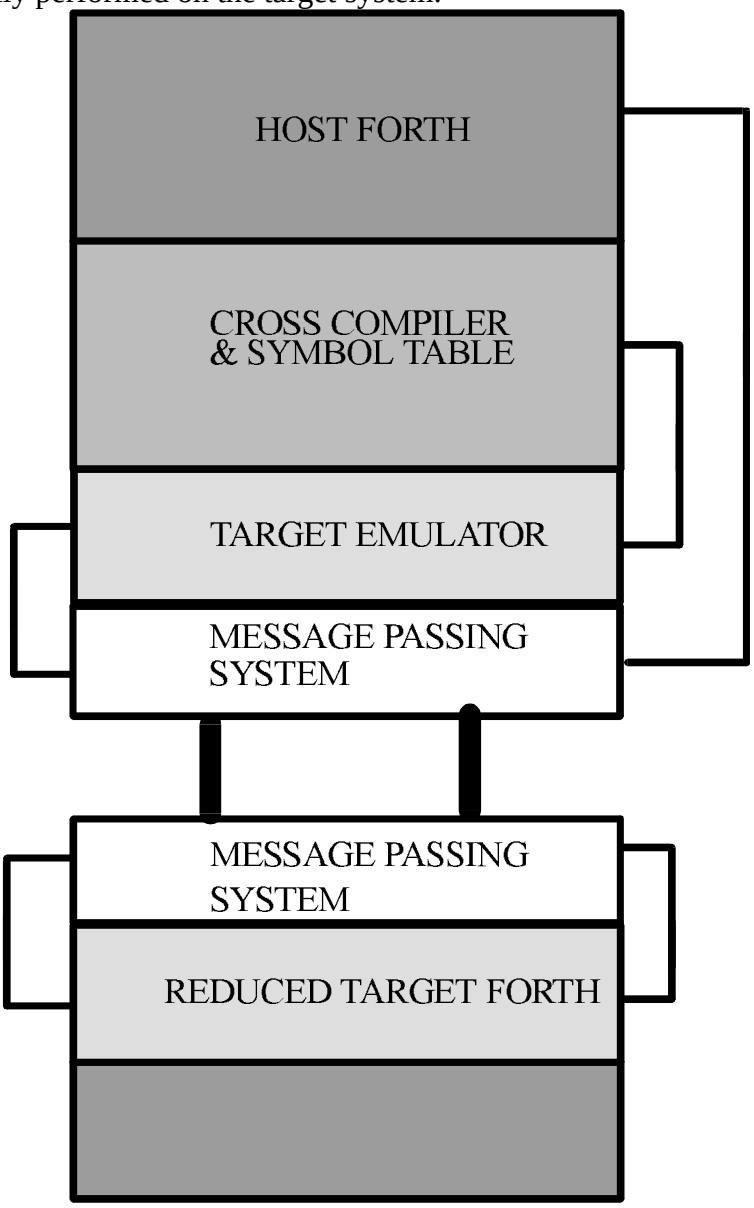
VARIABLE —
Available via the Umbilical Forth target emulator (on host).

WHILE —
Available via the Umbilical Forth target emulator (on host).

XOR n1 n2 — n3

7.5 Umbilical Forth Model

The diagram below shows the execution model of Umbilical Forth, illustrating how the symbol handling is performed on the PC, whereas the execution is actually performed on the target system.



1: Umbilical Forth Model

7.6 How it works

Umbilical Forth consists simply of a host section and target section of Forth which communicate via a message passing system. The message passing system exists partly on the target and partly on the host, with a communications (often serial) link to join them together.

7.6.1 Message Passing System

This is the Umbilical Forth “join”. It is simply two pieces of similiar code, one of which exists on the target and one on the host. The message passing system passes requests between host and target. The basic requests are to “pop from the target stack”, “push onto the target stack” and “execute a word on the target”.

7.6.2 Target Forth

This is a reduced word set Forth which runs on the target and makes requests of the host Forth via the message passing system.

7.6.3 Target Emulator

The target emulator exists on the host, its function being to do the target’s interpreting and compilation. It achieves this by accepting words from the host input stream, looking them up in the cross compiler’s symbol table and then either executing them or compiling them down onto the target via the message passing system. Since most of Forth itself consists of interpreting and compiling along with other formatted and non formatted I/O, by giving this burden to the host part of Umbilical Forth, the target part can be made very minimal indeed. Also the comiling words such as **DOES> IF ELSE ENDIF DO** and **LOOP** can be taken out of the target code and executed on the host part of Umbilical Forth. These compiling words will compile address of **?BRANCH BRANCH (DO) (LOOP)** etc into the target.

7.6.4 Which words are done when and by whom?

Since a word in Umbilical Forth can exist in two places, either on the host or the target, then a specific search order must be followed to find the correct definition of the word required.

For example, if you execute **COLD** from the keyboard, the target version of **COLD** will execute because there is no special case version of **COLD** within the host portion of Umbilical Forth. The target version of **COLD** makes no reference to the dictionary, and thus the cross compiler's symbol table space is not cleared. Unless there is a special case in the host, the target word will execute. The symbol table can only be manipulated by the host.

To clear symbols from the cross compiler use **FORGET**. There is a special case of **FORGET** on the host, and the host version can manipulate both the symbol table and the target dictionary pointer.

7.6.5 Search orders

For both interpreting and compiling the search orders listed are followed. If the criteria of a step is met then the rest of the list is not followed through.

INTERPRETING

- 1. Search the **SCI** (Special Case Interpreter) vocabulary on host and execute if word found.
- 2. Search the **INTERPRETER** vocabulary on host and execute if word found.
- 3. Search cross compiler symbol table on host and execute word on target if word found.
- 4. Try to convert word to a number and leave on target stack.
- 5. Give "word is undefined" error message.

COMPILING

- 1. Search the **SCC** (Special Case Compiler) vocabulary on host and execute if word found.
- 2. Search the **COMPILER** vocabulary on the host and execute if word found.
- 3. Search the cross compiler symbol table on the host and compile address on target if word found.
- 4. Try to convert word to a number and compile as a literal on the target.
- 5. Give "word is undefined" error message.

7.7 How to extend it

This section assumes that you have used the Umbilical Forth system already, and that you are familiar with its concepts. You should also have read the chapter on advanced concepts and extending the cross compiler.

Obviously, as a Forth, Umbilical Forth can have new definitions addded by the **:** and **;** words. These words will be added to the Forth on the target. However, some types of words can not exist on the target. Examples of this are defining words or any of the words which need to use the input stream. These words and other special cases mentioned below need to be added to the host part of Umbilical Forth.

Some special cases already exist on the host side of Umbilical Forth. **WORDS** and **BACK** are typical examples.

To add new definitions to the host side of Umbilical see the screen file EMULATOR.SCR, which contains source for words such as **WORDS** and **BACK**. Here is the definitions of a simple version of **WORDS**:

```
ONLY FORTH  ALSO C-C
ALSO UMBILICAL A LSO SCI DEFINITIONS

: WORDS
  CC/SI  CONTEXT(H) @ NFA+ CONTEXT !
  WORDS ;

ONLY FORTH DEFINITIONS
```

Notice that this definitions of **WORDS** (which is available to the Umbilical Forth Interpreter and Compiler via the vocabulary **SCI**) includes the normal Forth definition of **WORDS**, only changing the context vocabulary to the cross compiler symbol table. This is a good example of how, by taking advantage of the fact that the cross compiler uses the normal Forth system structures, extra words can be added to Umbilical Forth. The incantation **CC/SI** must be the first word (including comments) after the word's name. This incantation switches the search order to prevent confusion.

To be able to use any word in interpretive mode which doesn't exist on the target, you will have to add its definition to the vocabulary **SCI** on the host. The word **EMIT** is one such word. The host definition of **EMIT** will pop a value from the target stack and use this with the normal host Forth definition of **EMIT**.

```
ONLY FORTH ALSO C-C
ALSO UMBILICAL  ALSO SCI DEFINITIONS

: EMIT
```

```
  CC/SI
  POP-CMND>T >TARGET
  EMIT ;

ONLY FORTH DEFINIITIONS
```

7.8 Installing a new memory driver

Usually no installation is required. We have already installed the extensions onto the host system for the Leburg LePROM EPROM emulator. An example target implementation is also provided in the CHIP directory.

If you are not using the Leburg LePROM EPROM emulator, and wish to use a different emulator, you will have to write new emulator drivers. Note that the EPROM emulator is only necessary to modify the EPROM area, it does not affect interpretation, only compilation.

This section discusses the topic of writing new EPROM emulator drivers. Several examples are provided. These examples are taken from installationns for the Hitachi/Zilog 64180 and Hitachi HMCS400 processors. MPE can provide support if required.

This is the most difficult part of installing Umbilical Forth. To ensure that Umbilical Forth will compile onto the target board four words have to be written for the host side of the link. These words are **C!(U) !(U) C@(U)** and **@(U)**. Example EPROM emulator drivers for the LePROM may be found in the files EMULATOR.SCR (high level portion) in the UMBxxx directory and XC-EMU.SCR (low level portion) in the COMPILER directory. The source code of the cross compiler is provided in compressed form, and must be unpacked using the PKXARC utility.

7.8.1 Reading and Writing into Program Areas

The desired action of **C!(U)** and **!(U)** is to write to ROM/program space and of **C@(U)** and **@(U)** is to read from ROM/program space in the target. The stack effect of these words are:

```
C!(U)          \ b addr —
!(U) \ n addr —
C@(U)          \ addr — b
@(U)           \ addr — n
```

How the words are written to perform these functions is irrelevant. What is important is that the words you write will perform the actions given previously. Here are three different ways in which the host may have to communicate with the target:

- 1. Directly via the serial port on the target
- 2. Eprom emulator
- 3. Hardware emulator

The following examples will give some idea of how to write these words. The lower level portions are described and are called **O@(U)** and **O!(U)**.

7.8.2 Via the Target Serial Port

The Umbilical Forth prototype was developed using this method, the target processor being the Hitachi 64180. Since the 64180 has shared program and data spaces, the processor can write to any area of memory. This made it easy to provide both **O!(U)** and **O@(U)** because Umbilical Forth ran in RAM on the target. The way this was achieved was to use the message passing system to send an EXECUTE message to the target. The words then executed on the target were the normal RAM **!** and **@** Forth definitions. These are the definitions for **O!(U)** and **O@(U)**:

```
: O!(U)
  PUSH-CMND>T >TARGET
  PUSH-CMND>T >TARGET
  SYMBOL-OF ! TARGET-EXEC  ;

: O@(U)
  PUSH-CMND>T >TARGET
  SYMBOL-OF @ TARGET-EXEC
  POP-CMND>T >TARGET ;
```

The push and pop commands may seem to be initially confusing but they are needed because there are two data stacks in operation. The target emulator will leave and expect values on the host stack while the target words **!** and **@** executed on the target will need their values to be on the target stack. Hence the need to shuttle data between stacks.

7.8.3 Via an EPROM Emulator

The 64180 Umbilical Forth was then used with an EPROM emulator. In this case all that had changed was that the EPROM emulator took the place of RAM, but, it can obviously also be used to replace EPROM.

7.8.4 Hardware Processor Emulator

The most difficult case occurred when Umbilical Forth was implemented on a HMCS 400 emulator which in turn was front-ended by a 6301 program. The only way to communicate with the processor was by a 6800 monitor which could carry out the usual monitor functions such as modify memory and break-point.

Two major problems existed in the implementation of HMCS 400 Umbilical Forth:

1. The emulator is essentially non-interactive. The only way programs could be developed was to write code (assembler), down load this code to the emulator and run this under the monitor software, inserting break-points as appropriate to test the code.

2. Since all communication with the HMCS 400 processor had to be carried out via the 6800 emulator, how could the HMCS 400 target Forth and host Forth connect? How could the host part of Umbilical Forth compile onto the target?

7.9 Example solutions

The root cause of both problems discussed previously is the same. That is the fact that the host part of Umbilical Forth can only communicate with the 6800 emulator monitor, the HMCS 400 effectively ‘being in limbo’.

7.9.1 Interpret first

The first thing that should be attempted is just to bring Umbilical Forth up into a state where interpretation is possible from the keyboard. Worry about compilation later.

To give Umbilical Forth any hope of suceeding the message passing system has to be operational. This system relies on the host and target being able to communicate. To achieve this, both host and target parts of HMCS 400 Umbilical Forth communicate by exploiting the monitor.

1. Both host and target parts of the message passing system rely on the fact that the other part is sending if one is receiving and vice versa. That is the message passing system is a synchronous system.

2. The code on the HMCS 400 can be halted and control returned back to the monitor if an illegal opcode is encountered. The technique of using an illegal opcode is nearly always available if a processor emulator is being used.

Whenever the target part of Umbilical Forth needs to communicate with the host part, either expecting to receive a byte or attempting to send one, it executes an illegal opcode. At the same time the host part of Umbilical Forth is either waiting to send or receive a byte byte, and can do so by requesting that the now active monitor (brought about by the execution of an illegal opcode) read or write to target memory. By always reading and writing to the same address in target memory, bytes can be sent between the host and target parts of Umbilcal Forth. The host part of Umbilical Forth can request that the target resume running by telling the monitor to execute a “GO” command.

It can now be seen how, by the target executing illegal opcodes and then instructing the monitor to read or write target memory, both host and target parts of Umbilical Forth can send and receive bytes. This then is how the message passing system can be implemented and interaction and interpretation made possible.

7.9.2 Dealing with Compilation

Compilation relies on the same process as interpretation, that of implementing the message passing system using the illegal opcode and monitor method. It is different only in that the host part of Umbilical Forth will supply the byte and the dictionary pointer's position on the target to the monitor, which will then write to that address. The complication of writing into the HMCS 400's program space (which the processor itself can not do) is resolved because the monitor can do so through the emulator.

7.9.3 The words O@(U) and O!(U)

What this example should tell you is that the way to implement **O@(U)** and **O!(U)** on the hardware emulator method of host target communication can be DIRTY, CRUDE or IGENIOUS - anything that works is acceptable. Since many in-circuit emulators do not provide a program interface, only a user interface, processing the responses from an in-circuit emulator can be very tedious if short cuts are not taken.

Here are the HMCS 400 emulator versions of **O!(U)** and **O@(U)**:

```
: (A@)
  SERIAL-IO  CLEAR-INPUT
  ASCII I EMIT  SPACE  .HEX
  0D EMIT  REST
  02E EMIT  0D EMIT  LONG-REST
  CONSOLE-IO ;

: A@
  (A@)
  RING-BUFFER IN-INDEX @ + 0B -              \ dirty emulator
  3 OVER C!
  NUMBER? 0= ;

: A!
  SERIAL-IO  CLEAR-INPUT
  ASCII I EMIT  SPACE  .HEX
  0D EMIT  REST  REST
  0 # # # # # # TYPE
  02E EMIT  0D EMIT
  CONSOLE-IO ;

: O!(U)
  A! ;

: O@(U)
  A@ DUP -1 =
  ABORT" ERROR READING EMULATOR" ;
```

It can be seen that the **A!** and **A@** words accept input, tidy it, and deliver output to the monitor via the serial link.

7.9.4 Finally

Once the **O!(U)** and **O@(U)** words are written they are installed by ensuring that the word RESET-XC in the EMULATOR.SCR screen file contains the statements:

```
ASSIGN O!(U) TO-DO !(O)
ASSIGN O@(U) TO-DO @(O)
```

7.10 How to install the software

Assuming that the **Installing a new memory driver** section has been carried out correctly, then the actual installation of Umbilical Forth is quite straightforward.

7.10.1 Host

The host end of Umbilical Forth is easy to install. Regenerate the cross compiler in the SOURCE directory using the B.BAT batch file. This will perform the regeneration, and then deliver the required files into the compiler directory.

This has now generated the host end of Umbilical Forth.

Advanced Topics

8.1 Introduction

This chapter covers topics which are best considered when some familiarity with the use of the cross compiler has been obtained. These topics are not in themselves difficult, but require some understanding of Forth itself, and of the sequences of events that occur when the compiler operates.

8.2 Compiling for Minimum Size

There are three methods of generating minimum sized targets, and which one is used depends on the starting point of the application. The first method is to remove the dictionary headers of any words that are not needed by the text interpreter when the application runs. The second method is never to include unnecessary code, and the third is to remove unnecessary code. In turn unnecessary code exists either for interactive use of an open Forth system (the text interpreter and compiler layers), or for testing during development.

A fourth method, refactorisation, is actually a method of improving Forth code. A usual side effect of refactorisation is a reduction in code size.

8.2.1 Removing Dictionary Headers

If the target system is going to include a Forth interpreter, it is often likely that only a few words ever need to be interpreted. In practice open Forth systems on the target are usually used only for maintenance purposes, or to produce dedicated command scanners with only a small number of active words. In these cases, the dictionary headers of the other words are not needed. The compiler directives **INTERNAL** and **EXTERNAL** stop and restart respectively the production of dictionary headers. Sections of code bracketed by **INTERNAL** and **EXTERNAL** will thus contain no headers. Depending on the author's style, dictionary headers can account for between 20% and 30% of application dictionary use. The directive **NO-HEADS** can be used to prevent generation of dictionary headers regardless of the use of **INTERNAL** and **EXTERNAL**. After removing the dictionary headers completely, the interpreter and compiler layers can themselves be removed.

8.2.2 Umbilical Forth

Umbilical Forth was designed for the first approach, in particular for single chip applications. Since an Umbilical Forth application does not contain the Forth interpreter or compiler layers on the target, they need not be removed by the user as they are not supplied, but are simulated by the PC. Unnecessary code can be removed using the same trial compilation techniques as will be described for the second approach.

8.2.3 Removing Excess Code

The second approach involves removing unused code. This can be done manually, or with the help of the compiler. This is always a tedious chore, but correct use of the cross compiler makes it a mechanical process.

At the end of a compilation run the compiler produces a list of unresolved references. These are words that have been referenced but not defined. If you compile just the main word of an application, the compiler will issue a list of the words used by the main word. If you then add those words and recompile, another list will be produced. After several iterations there will then be no words in the list, and the process is complete.

In practice a mixture of manual pruning and use of the cross compiler is useful. For example, when removing the Forth interpreter and compiler layers all **IM-**

MEDIATE words can be removed. This set includes all the structure words such as:

IF ELSE ENDIF THEN BEGIN UNTIL WHILE REPEAT AGAIN CASE OF ENDOF ENDCASE DO ?DO LOOP +LOOP LEAVE ?LEAVE

Many target applications do not need the text input functions such as:

QUERY EXPECT

and some do not even require character i/o such as:

KEY KEY? EMIT TYPE SPACE SPACES

and consequently none of the number output words:

. U. D. .R U.R D.R

nor their factors:

#S # HOLD SIGN

If you are removing the interpreter and compiler layer, a good initial approach is comment out all references to WRK83.SCR in the control file except those references to compiler directives. None of the words in this file will be compiled, and the unresolved words list will tell you which ones to put back.

8.2.4 Refactorisation

During the development of an application many little phrases become commonplace in a number of words. These phrases can be extracted and replaced by one word, saving space. In turn the code will be easier to read, and this reveals further potential savings.

It is also possible to gain some space by converting between high level and assembler definitions, although there are no hard and fast rules for this, and the space saving is often marginal.

Refactorisation is also an important tool when trying to optimise the speed of an application as it often reveals high level words whose recoding into assembler has a significant impact on speed.

8.3 Defining Words

One of the major facilities in Forth is the defining word. Defining words allow the user to define a data structure and the way it is used in one definition. A defining word follows the forms:

```
: <name>
  CREATE   <compile time action>
  DOES>   <run time action> ;
```

```
: <name>
  CREATE  <compile time action>
 ;CODE  <run time action>
 END-CODE
```

When the parent defining word executes the data structure is built by the compile time action, and run time action is only used by the child definition when the child itself executes.

For the cross compiler to handle defining words automatically, it need only handle the compile time action, as the run-time action is only used by the target code. The cross compiler builds its own version of the compile time action into the cross compiler itself. When the defining word is used, it is the cross compiler's version that is executed.

Most of the time this action works without any intervention by the user. Because of the flexibility of Forth the compiler can sometimes be fooled. This chapter gives an overview of areas for which some caution is needed. For many users this will only be background information.

8.3.1 Using defining words

In the majority of cases the cross compiler deals with defining words automatically. The following code will be correctly compiled. The action of the word **DEFER** can be changed as required, and a typical use would be to create **EMIT TYPE KEY KEY?** and **CR** in this way if their action has to be changed by the application.

```
: CRASH                          \ — ; initial action
  1 ABORT" Unitialised deferred word"  ;
```

```
: DEFER                         \ —
  CREATE  [‘] CRASH ,           \ lay down initial action
  DOES   @ EXECUTE  ;           \ run time execute stored CFA

DEFER TEST1                     \ create a deferred word

‘ WORDS  ‘ TEST1 2+ !           \ store CFA of WORDS in PFA
```

Defining words which use **CREATE** ... **;CODE** are also handled correctly.

8.3.2 Defining words in ROM/RAM systems

The cross compiler contains some predefined host versions of defining words. In particular **VARIABLE** compiles different code depending on whether or not a ROM/RAM system is being generated. In this case two areas of target memory are being managed.

1. The ROM space, whose end is denoted by **HERE**, and which is adjusted by **ALLOT**, **C,**, and **,**.

2. The RAM space, whose end is denoted by **THERE**, and which is adjusted by **ALLOT-RAM**, **C,(R)**, and **,(R)**.

Defining words must be careful to use the correct version so that data is stored into the correct memory space, or that room is allotted in the correct space.

Regardless of which is given in the source code, the cross compiler will generate the TARGET words **HERE** and **ALLOT**. This is because when you compile in the target the code must be compiled into RAM. To illustrate the problems that can arise we will use **DEFER** as before. A first (and very reasonable) definition is given below.

```
: DEFER                                 \ —
  CREATE  THERE ,                       \ lay address
       2 ALLOT-RAM                      \ get space
      [‘] CRASH THERE !         \ initial action
  DOES>  @ @ EXECUTE  ;                 \ execute stored CFA

DEFER TEST1                             \ create word

‘ WORDS  ‘ TEST1 2+ @ !                 \ place CFA in RAM
```

This version fails when executed in the target because when the CFA of **CRASH** is stored the value of **THERE** (compiled as **HERE**) has been modified by the **ALLOT-RAM** (compiled as **ALLOT**). A more successful version is given below.

```
: DEFER                                 \ —
  CREATE  THERE ,                       \ lay address
       [‘] CRASH THERE !         \ initial action
       2 ALLOT-RAM                      \ reserve space
  DOES>  @ @ EXECUTE  ;                 \ execute stored CFA
```

The simplest route round is to use **,(R)** which reserves space in the RAM area, and is compiled as **,** in the target:

```
: DEFER                                  \ —
  CREATE  THERE ,                        \ lay address
       [‘] CRASH ,(R)            \ initial action
  DOES>  @ @ EXECUTE  ;                  \ execute stored CFA
```

The only secret to correct operation is to remember which space you actually want the data placed into. Even the last example fails in the target because **THERE** compiles **HERE** and the value required in the target is actuall **HERE 2+**. The best solution is to store addresses after reserving space, e.g:

```
: DEFER                                  \ —
  CREATE  0 ,                            \ reserve space
       THERE HERE 2- !           \ store address
       [‘] CRASH ,(R)            \ initial action
  DOES>  @ @ EXECUTE  ;                  \ execute stored CFA
```

The cross compiler contains special versions of defining words in many cases such as **VARIABLE** and **CONSTANT**. The source code for these will be found in the cross compiler core on the screens for the words in the **INTERPRETER** vocabulary. If such a special case exists, it will override the later version supplied in the target source code. Thus the target version is only used when the target executes, the action at cross-compilation being controlled by the cross-compiler version. The two versions must correspond by laying down data that is accessed by the run-time portion (after **DOES>**) supplied by the target code.

By compiling special versions into the cross compiler, the constraints demonstrated above are removed. These constraints are caused by the different memory layouts involved in cross-compilation and target execution - the target cannot compile into ROM. The removal of these restrictions allows the definition of **DEFER** above to reduce to the expected (and more readable) version below, at the expense of having to create a special version in the cross compiler. Many examples are to be found in the cross-compiler core source code.

```
: DEFER                                  \ —
  CREATE  HERE 2+ ,                      \ lay address
       [‘] CRASH ,               \ initial action
  DOES>  @ @ EXECUTE  ;                  \ execute stored CFA
```

This version lays data in a form that allows the run-time portion after **DOES** to handle data laid down by either the **DEFER** in the cross compiler or this last **DEFER**. A later section has more details on extending the cross compiler.

8.3.3 Variables between CREATE ... DOES>

When using defining words to create structures or lists it is often useful to refer to the contents of a variable. For example, a chain is often built as follows:

```
VARIABLE CHAIN-LINK

: defining-word                 \
  CREATE  ......                \ compile-time actionm
        HERE                    \ new link value
        CHAIN-LINK @ ,          \ lay old value
        CHAIN-LINK !                        \ update link var.
        ......                  \
  DOES>  ......  ;              \ run-time action
```

When the cross compiler encounters a target variable in a **CREATE** ... **DOES>** or **CREATE** ... **;CODE** structure, a reference to the target variable's RAM location is compiled. This, together with the modified memory operators which refer to target memory space, ensures that the value of the variable is correctly accessed and updated. In turn, the initial value laid down by the cross compiler will also be correct.

8.3.4 Restrictions on defining words

There are a few restrictions on the use of defining words, and these arise from the nature of the compiler. As the compiler may be geneating code for a different processor, it cannot execute the target code. When a defining word is encountered (identified by **CREATE**), the subsequent code up to **DOES** or **;CODE** is compiled into the host as well as into the target. Words such as **@** and **!** compile special forms into the host so that when the host version executes it executes correctly.

1. **CREATE** must be the first word after the name. This is because **CREATE** is the switch that enables an analogue to be built into the host.

2. No forward referencing is permitted between **CREATE** and **DOES** or **;CODE**.

3. The words between **CREATE** and **DOES>** or **;CODE** must exist in the host's **FORTH** vocabulary, except for variables and constants. The host word must also have the same action as the target word. These restrictions ensure that the host executes a correct version of the word. Variables and constants are permitted after **CREATE** to allow the construction of record-defining words. If necessary, additional words can be added to the host before **CROSS-COMPILE** is executed.

If you cannot meet these restrictions the defining word will have to be manually built into the cross compiler. If the word need not be executed in the host bracket it with **TARGET-ONLY** and **HOST&TARGET** so that no host version is built. If a defining word is to be compiled, and a host version already exists, a new version will NOT be compiled. This allows words such as **:** and **CONSTANT** to have predefined actions.

8.3.5 Words affected in defining words

To ensure correct execution of defining words some words compile special versions in the host version of the defining word. For instance, during execution of the defining word, the word **@** must refer to target addresses, not host addresses. The following words have been modfied to allow use inside defining words and immediate words.

DO ?DO LOOP +LOOP IF ELSE THEN ENDIF , C, @ C@ ! C! ." .(
CASE OF ENDOF ENDCASE HERE THERE ALLOT WORD

Other words may have been added since this manual was written. Such words may be identified within the cross compiler source listing. They are in the **COMPILER** vocabulary and contain the phrase:

?B-D IF ..... ENDIF

which contains the special code executed if inside a defining or immediate word. If the word is not taken as a special case, the host's **FORTH** vocabulary version will be executed by the defining word.

8.3.6 Adding defining words to the compiler

New defining words can be added the compiler very easily. Defining words are added to the cross compiler's **INTERPRETER** vocabulary, which contains words with special actions during interpretation. Some standard defining words, such as **VARIABLE :** and **USER** already have analogues in the host. Provided that such analogues can use the TARGET's run-time action, correct code will be generated. This topic is considered in much greater detail in a later section of this chapter.

8.4

Immediate words

As with defining words, it is useful for the cross compiler to be able to execute immediate words defined for the target. Tagging the words with the word **IMMEDIATE** is not sufficient as this only marks them in the target code.

If the word **I:** is used instead of **:** an analogue of a target will be built into the cross compiler's **COMPILER** vocabulary. To be immediate in the target it must still be tagged by **IMMEDIATE** in the usual way. The rules are the same as for automatic creation of defining words.

8.5 Extending the compiler

There are some occasions when it is useful to be able to extend or modify the cross compiler. The ability to extend the cross compiler itself gives the user enourmous power, but care is needed when testing. The usual reason the compiler is extended is to compe with a defining word or an immediate word that the cross compiler itself cannot cope with automatically.

Another reason to add words to the cross compiler is to add a compiling word which is only used during cross compilation and is unused when the target executes. Such a word need not be compiled into the target at all. Examples of such words are the **CASE OF ENDOF ENDCASE** words which compile branches. Another class is found in state-smart words. State smart words have to be immediate so that their action can be different in the interpreting and compiling states. The compiler cannot cope with such words automatically and they must be explicitly added.

8.5.1 How to Extend the Compiler

Study the cross compiler core listing, and then back everything up before starting! The cross compiler is quite a large application in itself, and deserves some attention to get the best out of modifying or extending it. Normally, the only source file of interest during modification is XDTC41.SCR, which contains the code that the user comes into contact with most of the time. The other files provide most of the housekeeping for the cross compiler. The Forth-83 nucleus supplied for rebuilding the compiler is called COMPCORE.COM. It is compatible with the target code, and is a version of PowerForth. The cross compiler can be extended in three ways.

- 1. Compile in new definitions before **CROSS-COMPILE** is executed. This allows the compiler extension code to be kept with the relevant target code. Unless major surgery is to be performed this is the recommended process.

- 2. Edit the cross compiler source code, rebuild the compiler on top of the supplied nucleus, and then test it.
- 3. Use **HOST-COMPILATION** and **TARGET-COMPILATION** to temporarily switch off the cross compiler.

Method 1
This is the preferred method in the vast majority of cases. It is simple to use, and the compiler extensions can be collected together in one file. This technique is used to load the compiler extensions for the 'large' multi-tasker supplied with some targets.

The extensions are loaded before **CROSS-COMPILE** is executed. The fragment below selects the **COMPILER** and **INTERPRETER** vocabularies in turn, and then restores the search order.

```
ONLY FORTH  ALSO C-C        \ access to cross compiler
CC/C                        \ select COMPILER definitions
.....
CC/I                        \ select INTERPRETER definitions
.....
ONLY FORTH DEFINITIONS      \ restore search order
```

Method 2
Editing and rebuilding the compiler is only recommended for major surgery.

Method 3
The words **HOST-COMPILATION** and **TARGET-COMPILATION** allow the cross-compiling action to be switched off for a while and then resumed. **HOST-COMPILATION** is the only way the cross compiler can be turned off once **CROSS-COMPILE** has been executed. Many users will not need this feature at all, and it is not often used as cross compiler extensions are most often compiled in before **CROSS-COMPILE** is executed. However the same techniques of word construction are applicable in either case.

8.5.2 Example compiler extension

The defining word **DEFER** is often used to create a word whose action can be changed later, e.g.

```
DEFER TEST1
```

The action is then assigned by the words **ASSIGN** and **TO-DO** as follows:

```
ASSIGN WORDS TO-DO TEST1
```

so that when **TEST1** executes, it performs **WORDS**. For the sake of convenience **ASSIGN** and **TO-DO** are state-smart words. When they are used inside colon definitions the assignation happens when that word executes, for example:

```
: SET-UP
  ASSIGN WORDS TO-DO TEST1  ;
```

The words **DEFER ASSIGN** and **TO-DO** have already been added to the compiler, and the example below describes the techniques involved. They are described here as an example of the relationship between normal Forth words, and their cross compiler equivalents. In this particular example the code is given for **ASSIGN** and **TO-DO** in the target. The target code for these words is only required for interpreting from the keyboard, and not for applications use, when **ASSIGN** and **TO-DO** may be omitted, leaving only **(TO-DO)** to handle the run-time actions.

To achieve the state-smart action, two words have to be defined in the cross compiler. The interpret-time action is placed in the **INTERPRETER** vocabulary, and the compile-time action is placed in the **COMPILER** vocabulary. A target word is referred to by using the words **TARGET** or **SEARCH**, and a reference to it is compiled by the word **COMPILE-REF**, while the CFA is found by the word **REF**. The example below assumes that the words **CRASH** and **DEFER** have already been defined in the target. The words ending in **(T)** are cross compiler analogues of normal Forth words, but the addresses are in the target system. The example below defines the target words first, and then switches the cross compiler off while the cross compiler words are defined.

```
: ASSIGN                        \ compile or return CFA
  STATE @                       \ must be state smart
  IF  [COMPILE] [']             \ compile time action
  ELSE  '                       \ interpret action
  ENDIF  ;  IMMEDIATE           \ must be immediate

: (TO-DO)                       \ cfa —
  R DUP 2+ R                    \ skip past compiled word
  @ 2+                          \ pick CFA, convert to PFA
  @                             \ pick up address in RAM
  ! ;                \ store CFA of action in RAM
```

```
: TO-DO                         \ compile (TO-DO) or act
  STATE @                       \ state smart again
  IF  COMPILE (TO-DO)           \ lay execution action
  ELSE  ' BODY @ !              \ store CFA at addr in PFA
  ENDIF  ;  IMMEDIATE           \ must be immediate


HOST-COMPILATION                \ switch cross compiler off
ONLY FORTH ALSO C-C             \ set up vocabularies

CC/I                            \ definitions in INTERPRETER
: ASSIGN        CC/I            \ — cfa
                '(T)  ;         \ synonym for '

: TO-DO         CC/I            \ cfa —
                '(T)            \ get target CFA
                2+ @(T)         \ get address from PFA
                !(T)  ;         \ store CFA in RAM address

CC/C                            \ next definitions in COMPILER
: ASSIGN        CC/C            \
  ['(T)]  ;                     \ synonym for ['] 

: TO-DO         CC/C            \ —
  TARGET (TO-DO)                \ get reference to target word
  COMPILE-REF  ;\ compile reference to (TO-DO)

ONLY FORTH                      \ reset vocabularies
ALSO C-C DEFINITIONS            \
TARGET-COMPILATION              \ switch cross compiler back on
```

For details of some of the cross compiler words consult the cross compiler core listing. The cross compiler's **INTERPRETER** and **COMPILER** vocabularies contain many relevant examples.

New words are usually added to the cross compiler's **INTERPRETER** and **COMPILER** vocabularies, since these vocabularies contain the cross compilers available special words for the interpreting and compiling states. Unlike a normal Forth system, the cross compiler has a third state, assembling. The words available during assembly are in the **ASM-ACCESS** vocabulary.

8.6

Cross Compiler Mechanics

8.6.1 The words TARGET and SEARCH

Before **CROSS-COMPILE** has executed, or once **HOST-COMPILATION** has been executed, the cross compiler is switched off. References to target words are no longer automatic. Thus words built into the cross compiler at this stage must use a special way of referring to target words.

A target word is referred to by using the words **TARGET** or **SEARCH**, and a reference to it is compiled by the word **COMPILE-REF**, while the CFA is found by the word **REF**. Both **TARGET** and **SEARCH** return the address in the symbol table of the word referred to. **REF** converts a symbol table address to the address of the CFA in the target (forward referencing is permitted). **COMPILE-REF** compiles a reference to the Forth word whose symbol table address is supplied (forward referencing is permitted).

Thus to refer to compile the target version of **?BRANCH**, a coss compiler word will use the phrase:

... TARGET ?BRANCH COMPILE-REF ...

The words referred to by **TARGET** must have been defined when the cross compiler itself was built. Normally the target symbol table is built when the cross compiler runs, so there is a mechanism that allows **TARGET** to refer to symbols that the cross compiler uses internally.

Thus, if the user needs to refer to a word that the user has defined, another mechanism must be used, and this is serviced by the word **SEARCH**.

```
: NEEDED                    \ —
  .......   ;

HOST-COMPILATION

.....
: MY-DEFINING-WORD          \ —
  .......
  SEARCH NEEDED COMPILE-REF
  .......  ;
```

TARGET-COMPILATION

8.6.2 Accessing Target Memory

Once the compiler has been turned off target memory is accessed by analogues of the normal Forth memory operatiors, bu siffixed with **(T)**. Thus target byte accesses are performed by **C@(T)** and **C!(T)**.

C@(T) C!(T) @(T) !(T) +!(T)

C,(R) ,(R)

HERE(T) LATEST(T) C,(T) ,(T) ALLOT(T) ALLOT-RAM

In Harvard targets (split CODE/DATA) such as the 8031, there will be additional words to access the various memory spaces.

8.6.3 Browsing supplied source code

The relevant part of the cross compiler source code is the file XDTC41H.SCR, which contains many examples of predefined defining and immediate words in its later sections.

8.6.4 Rebuilding the cross compiler

Unpack the SOURCE directory using PKUNZIP or PKARC, depending on whether the compressed file has a .ZIP or .ARC extension. Regeneration of the cross compiler is performed by the B.BAT batch file, which should be run from withinn the SOURCE directory. After the compiler has been generated, the required files will be copied into the COMPILER directory for execution. Once the compiler has been successfully modified, the SOURCE directory can be compressed again using PKZIP or PKARC.

Please note that all future releases will be issued using ZIP files for compression.

Compiler Directives Glossary

This glossary details the cross compiler directives. They are laid out in the usual fashion with stack comments and a description of their action.

;P —

End Page. Causes **FROM** and **FROM-FILE** to stop using the current page. Has the same effect as **;S** in screen files.

```
;P
```

(—
"open-paren"

The comment in a text differs from the screen file version. The paranthesis comment works over multiple lines. The end of comment is marked by a white-space-delimited closing parenthesis, e.g.

```
... 2DUP ( save adr # for later )  INIT ...
    ...
CLOBBER  ( NOTE:
                    use the operating system function
                    here to shut off access
         )
        ...
```

)ELSE(— I
"paren-else-paren"

The words **IF( )ELSE( )ENDIF** permit conditional compilation and interpretation. They are an analogue of **IF ELSE ENDIF** and are used in the forms:

```
flag
IF(  <if flag is true>  )ENDIF
```

```
flag
IF(  <if flag is true>
)ELSE(  <if flag is false>
)ENDIF
```

If the flag is true the words after **IF(** are interpreted or compiled, but if the flag is false the words after **)ELSE(** are interpreted/compiled. The **)ELSE(** clause is optional.

)ENDIF — I
"paren-endif"

See **IF(** and **)ELSE(**.

ALIAS —
"alias"

A defining word used to create a second symbol name for a word. Used in the form:

ALIAS <old-name> <new-name>
ALIAS (EMIT) SEND-BYTE

This feature is useful where a package is written using one word name, and the required function can be provided by a word of a different name elsewhere in the system. In the example above the CFA of **(EMIT)** will be compiled for a reference to **SEND-BYTE**.

ALIGN —
"align"

Used if the target processor requires 16-bit items to be aligned on a word boundary. This is sometimes a requirement of 16-bit targets such as the 68000 or 8096. This directive forces the cross compiler to align the link field and code field on even addresses. Some processors (such as the 8096) need the CFA to be aligned on an odd address. In this case use **ALIGN-ODD**.

ALIGN

ALIGN-EVEN —
"align-even"

A synonym for **ALIGN**.

ALIGN-EVEN

ALIGN-ODD —
"align-odd"

Used for processors such as the 8096 family which need the CFAs to be on odd addresses (so that PFAs are on even ones). The link fields are still forced to a even address.

ALIGN-ODD

ALL — 1 -1

Used before **FROM**, **DISPLAY**, or **CONTENTS**, **ALL** will put the first and last page numbers on the stack so that ALL the pages are specified. e.g.

ALL DISPLAY MYFILE.4TH

ALLOT-RAM n —
"allot-ram"

Allots space in the RAM area of a ROM/RAM target. This word is not strictly a compiler directive, but a modification of the normal Forth word **ALLOT**. Its function is to allow uninitialised space to be reserved in RAM.

```
THERE CONSTANT POINTER  \ pointer to RAM area
 0100 ALLOT-RAM         \ reserve space in RAM
```

ALONE n — n n
"alone"

A modifier used before **FROM** or **FROM-FILE** or other text file words to describe a single page, for example to compile page 5 from the current text file use:

5 ALONE FROM

BASE-36 —
"base-thirty-six"

Sets the current base to 36 (decimal). This apparently strange operation allows upper-case characters and digits to be encoded as radix-36 numbers, reducing the memory required. This technique is inherited from earlier incarnations of the compiler, where it was often used to encode the processor type as a double number, e.g.

```
BASE-36
M68HC11,  , ,                    \ lay as double number
DECIMAL
```

BOUNDING size mask end-gap start-skip —
"bounding"

This word is used in a ROM/RAM system to skip parts of memory that may be occupied by existing ROMS. Such a situation is sometimes found in domestic computers, or when modifying systems that use partial memory decoding.

n1 n2 n3 n4 BOUNDING

The first parameter n1 declares the size of the EPROM in bytes.

Parameter n2 is a bit mask identifying which address lines select the EPROM. For example an 8k EPROM of size 02000 (hex) is controlled by address lines A15,14,13 and will have a mask value of 0E000 (hex).

Parameter n3 specifies how far before the end of the EPROM the compiler stops to ask the user whether it should step to the next EPROM. This limit is used to prevent the last word in an EPROM being compiled across an EPROM boundary. If the answer is no, the qustion will be repeated before each word is compiled until the answer is yes or the EPROM boundary is crossed. When the yes answer is given, the compiler steps to the start of the next EPROM, and repeats the question. If the answer is again yes, the complete EPROM is skipped. Note that specifying a value of say 0100 does not guarantee to reserve 0100 free bytes at the end of the EPROM, it only specifies at what point the compiler starts asking what needs to be done.

Parameter n4 specifies how many bytes are skipped at the start of each EPROM.

The example below is for an 8k byte EPROM such that the compiler starts checking 128 bytes (080h) before the end of the EPROM, and the first 256 bytes (0100h) are skipped at the start of each EPROM.

HEX 02000 0E000 080 0100 BOUNDING

COMPILE-PAGE Page-Addr — ; <FileName>
"compile-page"

This word will cross compile a page that is executable from address PAGE-ADDR, the resulting image file then stored to disc as <FILENAME>.

```
HEX
08000 COMPILE-PAGE C:\PAGES\DRIVERS
```

CON: —
"to-con"

Direct the symbol table log output to the screen. The default output device. If required this directive has to be used before the command **CROSS-COMPILE**.

```
CON:
```

CROSS-COMPILE —
"cross-compile"

This word tells the system to start cross compiling. Until this point the compiler is a conventional Forth system with an application (the cross compiler) loaded but not running. At this stage extensions and assembler macros can be loaded. After **CROSS-COMPILE** has executed the compiler 'pulls down the shutters' to seal itself off, and then treats all code as target code and compiler directives.

```
CROSS-COMPILE
```

EMU-BASE address —
"emu-base"

Sets the base address of the LePROM emulator for the cross compiler.

```
HEX  0320 EMU-BASE          \ i/o address is 0320h
```

EQU n — ; <name>
"equ"

Creates a compiler-time equate of value n. When the equate is referred to in the target code the value assigned to the equate is used as a literal. An equate only exists during the run-time of the cross compiler. Equates may be redefined like macro-assembler set-symbols. Equates may be used wherever numbers may be used, they are just a means of naming a number.

```
HEX
0FF80 EQU IO-PORT
```

EXTERNAL —
"external"

The following words are generated with headers containing up to 31 characters, provided that the directive **NO-HEADS** has not been used.

```
EXTERNAL
```

FILE: — ; <filename>
"to-file"

Direct the symbol table log output to the file FileName. If required this directive has to be used before the command **CROSS-COMPILE**.

FILE: SYMBOLS.LOG

FINIS —
"finis"

This word stops the cross compiler. A cleaning-up operation is performed, final reports issued, the output file closed, and finally the compiler exits to the operating system.

FINIS

FINIS-PAGE —
"finis-page"

This word is used to finish off the compilation of a page, and return to cross compiling the kernel.

```
HEX
08000 COMPILE-PAGE PAGEA.IMG

ALSO PAGEA-VOC DEFINITIONS

3 LOAD

FINIS-PAGE

PREVIOUS DEFINITIONS
DECIMAL
```

Notice that before the source code is cross compiled with the phrase:

3 LOAD

the page's vocabulary has to be set as the definition vocabulary with the phrase:

ALSO PAGEA-VOC DEFINITIONS

This is essential and allows the page to be available interactively to the target.

FROM first last —
"from"

Get text input from a range of pages in the current file. The first and last pages to be used are supplied on the stack. Files can be nested to any depth; that is files can include input from other files. **FROM** is like **THRU** or **THRU-USING** with screen files, e.g.

4 10 FROM

FROM-FILE first last — ; <filename>
"from-file"

Get text input from a range of pages in the file <file-name> typed after **FROM-FILE**. The first and last pages to be used are supplied on the stack. Files can be nested; that is files can include input from other files. **FROM-FILE** is like **THRU** or **THRU-USING**, e.g.

4 10 FROM-FILE MYFILE.FTH

HEADS? — t/f I
"heads-query"

An immediate equate that returns false if the **NO-HEADS** directive has been used. The function of this equate is to return a value for condition compilation of the interpreter and compiler layers if heads are needed, for example:

HEADS?
IF(7 LOAD-USING INTERACTIVE)ENDIF

HOST&TARGET —
"host-and-target"

Used after **TARGET-ONLY** to allow defining words to be handled again. Some special cases of defining words cannot be handled by the cross compiler, but are required for target execution. **TARGET-ONLY** and **HOST&TARGET** handle this situation.

HOST&TARGET

HOST-COMPILATION —
"host-compilation"

Temporarily turns off the cross compiler so that the following code can be compiled into the cross compiler itself. Cross compilation is restarted by **TARGET-COMPILATION**.

HOST-COMPILATION
.....
TARGET-COMPILATION

I: — ; <wordname>
"i-colon"

Not really a directive, rather an auxiliary version of **:**. **I:** is used instead of **:** to create an immediate word that will exist in, and can be executed by, the cross compiler in the same way that defining words are handled. The same rules as for defining words apply to words created by **I:**.

I: <name>
.....
; IMMEDIATE

Note that **IMMEDIATE** only affects the target code, **I:** is needed to enable the analogue to be built into the cross compiler.

IF(n — I
"if-paren"

The words **IF( )ELSE( )ENDIF** permit conditional compilation and interpretation. They are an analogue of **IF ELSE ENDIF** and are used in the forms:

```
flag
IF( <if flag is true> )ENDIF
```

```
flag
IF( <if flag is true>
)ELSE( <if flag is false>
)ENDIF
```

If the flag is true the words after **IF(** are interpreted or compiled, but if the flag is false the words after **)ELSE(** are interpreted/compiled. The **)ELSE(** clause is optional.

INITIALISED-RAM #bytes —
"initialised-ram"

Changes the maximum size of the cross compiler's initial value table for ROM/RAM systems. This directive must be used before **ROM-BASE** or **RAM-BASE** are used.

```
DECIMAL 1024 INITIALISED-RAM
```

INTERNAL —
"internal"

This directive causes the following words to be generated without headers. The cross compiler still knows they are there, but they will not be visible to the compiled system.

```
INTERNAL
```

IS-FENCE —
"is-fence"

Marks the last word defined as being the last word in the protected dictionary. In a RAM system, the cross compiler places the NFA of this word at the location **INIT-FENCE**. The target start-up code can then access this location to find the initial value of the variable **FENCE**.

```
L: INIT-FENCE  0 ,                          \ reserve space

.....

: FORTH-83  ;  IS-FENCE                     \ fills INIT-FENCE
```

L: — ; <word-name>
"ell-colon"

Creates a label with the address returned by **HERE**, i.e. the current location in the dictionary. Normally only used during interpretation, but can be used during compilation if surrounded by **[** and **]**.

```
L: DATA-SLOT  0 ,

: <word>
......
[ L: INSIDE-WORD ]
......
;
```

LABEL addr — ; <word-name>
"label"

Used during interpretation to create a label at an arbitrary location to satisfy a forward reference from a code definition or code fragment.

```
THERE LABEL MY-DATA
```

LOAD n —
"load"

The contents of screen n of the current screen file are compiled, e.g.

```
10 LOAD
```

LOAD-USING n — ; <pathname>
"load-using"

The contents of screen n of the given screen file are compiled. If no extension is given, the compiler will add the extension '.SCR', e.g.

```
10 LOAD-USING A:\ROM-IO     \ load from A:\ROM-IO.SCR
```

LOG —
"log"

Generate a full symbol table log.

```
LOG
```

MEM-BASE addr —
"mem-base"

Used to set the base address of memory for a RAM system, e.g.

```
HEX
0100 MEM-BASE
```

MEM-END addr —
"mem-end"

Used to set the upper address + 1 of available RAM for both RAM and ROM/RAM systems. Even if the target determines its own RAM allocation this directive should be used, so that the compiler error checking does not use silly limits. e.g. for a CP/M system whose executable file starts at 0100h:

```
HEX
0100 MEM-BASE
0E000 MEM-END
```

For a ROM-based system with ROM at 0000, and RAM from 08000 to 09FFF, the following sequence would be used:

```
HEX
0 ROM-BASE
08000 RAM-BASE
0A000 MEM-END
```

NO-HEADS —
"no-heads"

Disables generation of heads, overriding **EXTERNAL** and **TARGET-WIDTH** completely. This directive can be used when the application is complete to remove ALL heads from the system without having to go through the source code removing all occurences of **EXTERNAL** and **TARGET-WIDTH**.

```
NO-HEADS
```

NO-LOG —
"no-log"

Generate a reduced symbol table log.

```
NO-LOG
```

ONWARDS n — n -1
"onwards"

A modifier used before **FROM** or **FROM-FILE** or other text file words to use the text from a specified page to the end, for example to compile page 5 to the end from the current text file use:

```
5 ONWARDS FROM
```

ORG addr —
"org"

A directive used to set the dictionary pointer to the given address. This directive is particularly useful when positioning code at a specific address. For example, on processors such as the Z80 for which interrupts are seviced at a particular address, **ORG** is used to force the dictionary pointer to the entry point.

```
HEX
08 ORG
ISR008 JP
018 ORG
ISR018 JP
```

OUTPUT-EMULATOR eprom-type bus-width —
"output-emulator"

Defines the EPROM type and bus width of the EPROM emulator the compiler will use. This directive switches all target memory words to use the EPROM emulator drivers instead of the output file drivers. Predefined constants exist to define the EPROM type and bus width. The EPROM type is one of:

E2764 27128 E27256 E27512

and the bus-width is one of:

8BIT 16BIT 32BIT

OUTPUT-FILE — ; <pathname>
"output-file"

Sets the output file to be used for the target image. If no extension to the filename is given, the extension '.IMG' will be added. The file is not opened/created until the base address of the output has been set by **MEM-BASE** or **ROM-BASE**. Use in the form:

```
OUTPUT-FILE <pathname>
OUTPUT-FILE ROMZ80
```

PRN: —
"to-prin"

Direct the symbol table log output to the printer. If required this directive has to be used before the command **CROSS-COMPILE**.

```
PRN:
```

PTO —
"p-t-o"

Please Turn Over. This word causes **FROM** and **FROM-FILE** to stop using the current page and to start on the next. (The same effect as —> in screen files.)

```
PTO
```

RAM-BASE addr —
"ram-base"

Used to set the base address of the RAM in a ROM/RAM system, e.g.

```
HEX
00000 ROM-BASE
08000 RAM-BASE
0A000 MEM-END
```

RAM-END addr —
"ram-end"

A synonym for **MEM-END**.

```
HEX 0E000 MEM-END          \ Memory ends at 0DFFFh
```

RESTART — ; <FileName>
"restart"

Continue cross compilation from the position saved under the given file name. The cross compiler saved position files having been generated using the directive **SUSPEND** during an earlier cross compilation.

RESTART KERNEL

ROM-BASE addr —
"rom-base"

Used to give the base address of the ROM and informs the compiler that a ROM/RAM system is to be generated, e.g.

HEX
0500 ROM-BASE

SUSPEND — ; <FileName>
"suspend"

Stop the present cross compilation, saving cross compiler information to disc under the given filename, the cross compiler then returning to XSHELL (or any program that called the compiler). Note that the file name should NOT include an extension, the cross compiler supplyies its own. The directive **RESTART** is used to resume cross compilation later.

SUSPEND KERNEL

TARGET-COMPILATION —
"target-compilation"

Re-enable cross compilation after it has been turned off by **HOST-COMPILATION**.

TARGET-WIDTH n—
"target-width"

Sets the maximum number of characters in the name field to be n. A maximum of 31 characters is imposed by the compiler.

7 TARGET-WIDTH

THRU n1 n2 —
"through"

The contents of screens n1 to n2 inclusive of the current screen file are compiled, e.g.

DECIMAL 7 23 THRU

It is good practice to define the number base just before **LOAD** or **THRU** as 'other people' sometimes forget to restore the base at the end of a screen, or it might be house policy only to define the base where it matters. It this instance it does.

THRU-USING n1 n2 — ; <pathname>
"through-using"

The contents of screens n1 to n2 of the given screen file are compiled. If no extension is given, the compiler will add the extension '.SCR', e.g.

```
DECIMAL  10 16 THRU-USING EXAMPLE
```

USE — ; <text-file-name>
"use"

Set the default text file you wish to **USE** (like "**USING** xxx.scr" for screen files). The file defaults to FORTH.FTH.

```
USE TEXTFILE.FTH
```

Warranties, Support, and Copyright

We try to make our products as good as we possibly can. We support our products. If you find a bug in this product we will do our best to fix it. Please send us a disc with a piece of sample code and a paper listing of the problem, and let us know the serial number of your issue disc. We will then send you an updated disc when we have fixed the problem. Do contact us or your supplier first in case the problem has already been fixed.

Technical support is available from MPE during office hours, or via access to our conference on the CIX (Compulink Information eXchange) bulletin board system on 01 399 5252. The MPE conference also has a closed area for registered customers, so log on and leave mail for MPE with your serial number to register for this.

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 mulitple copy terms and site licensing.

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. If you do not want to tear out the form from the manual, please photocopy it and send us the copy. You will find the serial number of the system on the original issue disc.

Please note that code generated by the cross compiler may be distributed without royalty. 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.

If your application is sealed (the user can’t get at it and does not know it is written in Forth) there will be no licence problems.

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. These terms do not apply for systems whose open portion is only avail-

able for maintenance purposes, provided that neither the Forth nor the means of access to it is documented in the user documentation.

The OEM licence allows developers to licence MPE documentation sets, and to use the FILETRAN, BIN-UP, and HEX-UP communications tools for up-loading and downloading both source and compiled image files to and from a host computer. Object files are transferred by XMODEM protocol, or in Intel Hex format. Source code is controlled by XON/XOFF protocol.

If you have any doubts or queries please contact us and we will do our best to help you.

10.1 Software Registration Form

SEND TO:
MicroProcessor Engineering Limited
133 Hill Lane
Shirley
Southampton SO1 5AF
England
tel: (+44) 703 631441

10.1.1 Customer details

We need your name, company, department, address, phone number, etc, in fact enough details so that we can send you letters, and answer telephone enquiries if we've lost the scrap of paper we took your phone number down on when you phoned. Overseas customers - please include the country with your address!

Name:

Company:

Department:

Address:

Telephone number/extension:

10.1.2 **Package details**

We need enough detail to be able to send you an update disc.

Title: **MPE Forth Cross Compiler Core version 4.1**

Processor and computer type:

Operating system:

Disc format: (single/double sided...)

Serial number:

Purchase date:

Bought from:

Blank Page

Example Control File

The cross compiler control file shown here is taken from the 68HC11 cross compiler target code. It shows the use of a control file for a multi-tasking target, and uses many of the features covered in the chapter on 'Controlling the Compiler'. Since this example is taken from a 'live' file, the usual convention of putting all code in upper case has been abandoned for this example. This file has only been edited so that it will fit on the page using the Courier type used.

```
only forth definitions decimal
img.pcb pathname rom6811.img    \ output file name

CROSS-COMPILE

only forth definitions

\ calculate ram allocations
hex
8 equ #tasks        \ number of tasks, at least 1
                    \ each task needs 0100 bytes
                    \ this space is reserved first
#tasks 0100 * equ taskram \ space used for tasks
0100 equ intram           \ interrupt page

\ define ROM and RAM areas        sfp 23/11/88
hex
\ ROM for 68xx nearly always in high memory
0C000 ROM-BASE      \ ROM starts at 0E000

\ User areas: 0100h/task + 1 page for interrupts;
\ requiring 0900h 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
02000 EQU INIT-U0   \ equate for task area base
INIT-U0 taskram +
  EQU int-init-u0   \ interrupt page base
int-init-u0 intram +
   equ INIT-TIB     \ starts at task+0900
INIT-TIB 0100 +
  RAM-BASE          \ Vars & Dict at task+0A00

\ MEM-END defines the end of RAM+1
04000 MEM-END       \ RAM ends at 03FFF

\ system addresses and equates    sfp 23/11/88
HEX
```

```
\ The user area, PAD, and two stacks are in one page per task
\ 000:03F       User variables
\ 040           PAD     defined in WRK83.SCR
\ 080:0BF       Data stack
\ 0C0:0FF       Return stack

INIT-U0 0FF + EQU INIT-R0       \ top of R. stack
init-u0 0C0 + equ init-s0       \ top of P. stack

\ system equates .....  sfp 09/11/88
hex
( release numbers and character equates )
30 EQU MPE-REL  30 EQU MPE-VER
31 EQU USRVER   20 EQU ABL
0D EQU ACR      2E EQU ADOT
07 EQU ABELL     8 EQU BSIN
08 EQU BSOUT    0A EQU ALF
0C EQU FFEED
0 EQU FALSE     -1 EQU TRUE

( buffer definitions and sizing )
2 EQU DBH       0400 EQU KBBUF
2 EQU DBT
DBH KBBUF DBT + + EQU HDBT

\ zero page equates             sfp 23/11/88
hex
0FE equ UP      \ pointer to user area
0FC equ RP      \ return stack pointer
0FA equ curtask   \ pointer to TCB of curr. task
0F9 equ task#     \ current task number

\ I/O register eqautes          sfp 09/11/88
HEX
0102B equ BAUD
0102C equ SCCR1
0102D equ SCCR2
0102E equ SCSR
0102F equ TX
0102F equ RX




\ pull in the files             sfp 13/11/88

DECIMAL 10 15 THRU-USING WRK83    \ system set up
DECIMAL  2    LOAD-USING CODE6811 \ main code defs
DECIMAL 19 24 THRU-USING WRK83    \ Forth part 1
```

```
DECIMAL  1    LOAD-USING ROM-IO   \ i/o drivers
decimal 25 67 thru-using wrk83    \ Forth part 2
decimal  3 9  thru-using romtools \ extensions
decimal  2    load-using multi11  \ multi-tasker

\ clean up                        sfp 17/03/88
hex
ecld 0FFFE !        \ set cold start vector
decimal
FINIS
```

Blank Page

Cross Interpreter Word Set

This glossary lists all the words that are treated as special cases by the cross compiler during interpretation (not inside a colon a definition). This set of words includes many of the compiler directives, plus those words which need changes to work properly. For example the compiler must use special versions of **@** and **!** so that target memory is referred to rather than host memory. For details of the

!	n addr —
“”	— ; compile string upto “
“,	— ; compile string upto ”
‘	n —
(	— ; ignore string upto & including
)ELSE(	— ; ignore string upto & including)ENDIF
)ENDIF	—
+!	n addr —
.(	— ; display string up to)
:	— ; word-name
;S	—
ALIAS	— ; <old-name> <new-name>
ALLOT	n —
ALSO	—
ASCII	— c ; <character>
ASSEMBLER	—
ASSIGN	— cfa ; <wordname>

BASE-36	—
C!	c addr —
C,	c —
C@	addr — c
CMOVE	source dest len —
CODE	— ; word-name
CONSTANT	n — ; constant-name
CREATE	— ; word-name
CVARIABLE	— ; <word-name>
DEFER	— ; wordname
DEFINITIONS	—
DLITERAL	d —
EQU	n — ; equate-name
ERASE	addr len —
FILL	addr len char —
FORTH	—
FROM-FILE	first last — ; <pathname>
FROM	first last—
HERE	— addr
HOST&TARGET	—
HOST-COMPILATION	—
I:	— ; word-name
IF(	— ; ignore string upto & including **)ELSE(** or **)ENDIF**
IMMEDIATE	—

IS-FENCE	addr —
L:	— ; label-name
LABEL	addr — ; label-name
LATEST	—
LITERAL	n —
LOAD-USING	screen# — ; pathname
LOAD	n —
MAKE-ITEM	— ; “large” multi-tasker only
MAKE-LIST	— ; “large” multi-tasker only
ONLY	—
ORG	addr — ; set dictionary pointer
PREVIOUS	—
TARGET-ONLY	—
THERE	— ram-addr
THRU-USING	start-screen# end-screen# — ; pathname
THRU	first last —
TO-DO	— ; wordname
TOGGLE	addr n —
USER	n — ; user-name
VARIABLE	n — ; variable-name
VOCABULARY	— ; vocabulary-name
[‘]	— addr ; word-name
[COMPILE]	— ; word-name compiles cfa
[PREVIOUS]	—

[— ; switch to interpreting

\ — ; ignore rest of input line

] — ; switch to compiling

Cross Compiler Word Set

This glossary lists all the Forth words that the cross compiler treats as special case during compilation (inside a colon definition). There are two sets of words involved.

The first set consists of words which are normally **IMMEDIATE**. They require an analogue in the cross compiler in the same way that defining words do.

The second set is of words that whose behaviour has to be considered carefully in defining words (inside **CREATE** ... **DOES>** and **CREATE** ... **;CODE**) and immediate words. Words such as **@** and **!** will be executed when the defining word is executed. Such words cannot use the the normal Forth function as that refers to the host Forth rather than the target image. Consequently many words have special cases for use inside defining words.

!	—
“”	— ; <string> compile string upto & including “
(	— ; <string> ignore string upto & including)
)ELSE(	—
)ENDIF	—
+!	—
+LOOP	—
,	—
—>	—
."	— ; <string> compile string upto & including “
.(	— ; display following comment
:	—
;CODE	—
;	—

?DO	—
?LEAVE	—
ABORT"	— ; <string> compile string upto & including “
AGAIN	—
ALLOT-RAM	—
ALLOT	—
ASCII	— ; <string>
ASSIGN	— ; <word-name>
BEGIN	—
C!	—
C,	—
C@	—
CASE	—
CMOVE	—
CREATE	— ; <string>
DLITERAL	d —
DOES>	—
DO	—
ELSE	—
ENDCASE	—
ENDIF	—
ENDOF	—
END	—
ERASE	—

FILL	—
HERE	—
IF(	—
IF	—
LEAVE	—
LITERAL	n —
LOOP	—
OF	—
REPEAT	—
SMUDGE	—
THEN	—
THERE	—
TO-DO	— ; <word-name>
UNTIL	—
WHILE	—
WORD	—
[‘]	— ; <word-name>
[COMPILE]	— ; <word-name>
[	—
\	— ; ignore rest of line
]	—

Blank Page

Umbilical Interpreter Word Set

This glossary details the words that are special cases to the Umbilical Forth interpreter. The definitions of the these words are in EMULATOR.SCR and are in the **SCI** vocabulary.

!	n addr —
!C	n addr — ; Harvard architectures only
‘	— CFA
+!	n addr —
,	n —
.	n —
?	addr —
“@”	addr — n
“@C”	addr — n ; Harvard architectures only
ALL	first last —
ALLOT	— n
ALONE	n — n -1 ; on target stack
ASCII	— char
ASSIGN	— cfa
BACK	— ; back to cross compiler
BINARY	—
BLANK	addr len —
BYE	— ; leave system completely
C!	b addr —

C!C	b addr —
C,	b —
C@	addr — b
C@C	addr — b ; Harvard architectures only
CDUMP	addr len — ; Harvard architectures only
CONSTANT	n —
DECIMAL	—
DIR	—
DUMP	addr len —
ED	— ; obsolete from Xshell v2.50
EDIT	— ; obsolete from XShell v2.50
EQU	n — ; <word-name>
ERASE	addr len —
FILL	addr len char —
FROM	first last — ; <file-name>
FROM-FILE	first last — ; <file-name>
HERE	— addr
HEX	—
LABEL	addr — ; <label-name>
LATEST	—
LIST	n —
LOAD	n —
LOAD-USING	n1 n2 —
ONWARDS	n — 1 n

THERE	— ram-addr
THRU	first last —
THRU-USING	first last — ; <file-name>
TO-DO	cfa — ; <word-name>
U.	u —
UPTO	n — 1 n
USE	—
USING	— ; <file-name>
WORDS	—

Blank Page