MPE Pinc PowerForth

User Manual

MPE Pinc PowerForth
Version 1.00Beta

User Manual
January 1990

Rod Crawford

Andrew Waters

Stephen Pelc

MPE Pinc PowerForth

MicroProcessor Engineering Limited
133 Hill Lane
Southampton
U.K.
SO1 5AF

Tel: +44 (0)23 8063 1441
Fax: +44 (0)23 8033 9691

Introduction

1.0 How To Use This Documentation

READ THIS SECTION,
EVEN IF YOU DON'T READ ANYTHING ELSE.

Many people read the user manual only 'when all else fails'. If you are one of these people, at least read this section, it will tell you where to find the information when you need it.

1.1 Organization of the manual

Each chapter of the manual covers a separate subject. These are detailed below:

Installation	To learn how to install Pinc PowerForth, read the "Installation And Backup" chapter. This details the contents of the disk, backup procedures, how to compile and run Pinc PowerForth, setting up of screen drivers, and error messages.
Tutorial	Although not a complete course on how to program in Forth (see bibliography for a list of recommended books), this chapter is intended to get you started using Pinc PowerForth.
Editing	Read the chapter "Source Code And Editing" to find out how to use disk files for source code storage.
Operating System	This chapter deals with file i/o, and memory allocation, using the features provided by the operation system.
Vectoring	This chapter describes vectored execution using deferred words.
Applications	The chapter on application programming describes how to go about creating auto-starting turnkey systems. Useful information about the workings of Pinc PowerForth is contained in the chapter on internals.
Bibliography	A list of books we at MPE have read and recommend to other Forth programmers.

Errors If Pinc PowerForth responds with an error number/message, you should consult the chapter on “Error Messages”, for a more detailed explanation.

Glossary When you come across a word name you do not know, or have forgotten, you should look up its definition in a glossary.
A glossary is a Forth term for an alphabetically sorted list of words, with a description of the action of each one.

Licence Information regarding the distribution of application programs & source code, warranty, copyright, support, and the Software Registration Form.

1.2 Nomenclature

Forth words in the text appear in upper case bold, e.g. the word **SWAP** is the Forth word to swap two items on the stack.

Program code examples are shown in courier font, thus:

```
: test  ( — )   \ test word
          ." Hi There"
;
```

Throughout this manual, when you see <cr> you should press the ENTER key (sometimes labeled the RETURN key). It stands for Carriage Return - another name for the key.

When describing text that you must type in for various commands, this manual uses the convention that the name of the type of information needed is surrounded by angle brackets. For example, if you were required to type the command **INCLUDE-FILE** followed by the pathname of the file to be included, it would be written thus:

INCLUDE-FILE <pathname>

i.e. this does not mean that you type the character “<”, followed by the character “p”, followed by the character “a”,etc.

1.3 MPE Pinc PowerForth

MicroProcessor Engineering’s Pinc PowerForth is an interactive programming environment for Workstations. It can be used standalone to build complete applications or be integrated into existing applications to provide interactive test and debug facilities. This product forms the basis of MPE Workstation Cross Compiler Range. It is a Forth implementation based upon the Forth-83 Standard, with many extensions.

1.4 Background

Forth was originally developed by Charles Moore as a computer language that amplified his own productivity as a programmer. Its first incarnations were used to control astronomical telescopes. What has emerged after a number of years is a language that shares the expressiveness of other high level languages, with the speed of machine code. Furthermore, Forth is extensible and interactive - allowing you to develop programs in easy stages at the keyboard. As you compile new words into the system, these are tested and integrated, adding to the general vocabulary of the Forth language. User defined words are available to you just like the predefined ones. Forth is a language of great richness and power, that enables a programmer to produce sophisticated programs quickly.

1.5 Features Of Pinc PowerForth

Pinc PowerForth is designed to operate within a workstation environment e.g. running under Unix or VMS. It is supplied in source form and can be compiled on workstations which have a C compiler conforming to either the K&R or proposed ANSI standards. Additional features of this product are:

- Source code can be edited using a standard text editor.
- Pinc PowerForth has been implemented in such a way that the operation of some i/o (input/output) dependent words can be controlled by the user. This enables additional devices to be used through the Forth i/o structure as well as through the operating system. The user can also by pass commands to the operating system or create an operating system shell.
- All disc input and output is performed through standard system calls.
- Vectored execution using deferred words - most i/o words supplied are deferred.

1.6 The Manual

This manual does not pretend to be a ‘teach yourself’ Forth or C text. It is a reference for this implementation. The “Tutorial” chapter for the newcomer to Forth is only intended to keep you going until you have been to a bookshop. The newcomer is strongly recommended to buy one of the books in the bibliography. These and many other Forth books are also available from MicroProcessor Engineering in Europe, or from Mountain View Press or the Forth Interest Group in the USA.

Installation And Backup

2.0 Introduction

Pinc PowerForth is normally supplied on an IBM/PC compatible 360k floppy disc containing several files. You should copy the files contained on this disk onto your workstations file system. (If you are unable to perform this task MPE can recommend disk copying services.) Beware of differences between line terminating characters (DOS uses <CR><LF> whereas UNIX uses <LF>. Before you do anything else, backup the contents of the supplied disc with your standard operating system file backup procedure. The master (supplied) disc is then only used in dire emergency, and **NOT** to make the software. If the issue disc is not write protected, please write protect it now before anything 'embarrassing' can happen.

2.1 Contents Of The Supplied Disc

The supplied disc will contain at least the following files:

release.doc	Updates produced between revisions of the manual will be noted in this file.
***.c**	C Source Code for Pinc PowerForth.
***.h**	Header Files for Pinc PowerForth.
cscape	Pinc PowerForth Secondary Code and Programming Environment.
***.mk**	Makefile for different workstation. e.g. **sun.mk** for the a sun workstation.

The names and contents of the .C files and .H files are detailed in the C Source Code Manual.

2.2 Installing Pinc PowerForth on a Supported Workstation.

This section describes how to install Pinc PowerForth on workstations that is supported by MPE. If you wish to install Pinc PowerForth on a workstation which not supported we recommend that you read this section **and** the section entitled "Installing Pinc PowerForth on Unsupported Workstation".

2.2.1 Selecting The Makefile.

Pinc PowerForth is generated via a makefile. Several of these are supplied, one for each workstation that MPE supports. Each file is terminated with the suffix **.mk**. Choose the one you require for you workstation and name it appropriately (some

make utilities require the makefile to be physically called makefile). E.g. If you are working with a Sun you should rename the file **sun.mk** to be **makefile**:

mv sun.mk makefile

2.2.2 Changing The Header Files.

Before making Pinc PowerForth it will be necessary to specify which implementation you intend to create. This should be done by editing the file **cmodel.h**. E.g. If you are implementing on a Sun workstation you would uncomment the line:

#define IMPLEMENTATION SUN

and comment out all other definitions of IMPLEMENTATION.

2.2.3 Making Pinc PowerForth.

To generate a Pinc PowerForth system you should run systems Make utility with the makefile that you chose in the above section. The complete make will take several minutes and will generate an executable version of Pinc PowerForth. That file is called: **pinc**.

You can execute Pinc PowerForth by typing in that name at the terminal.

2.3 Installing Pinc PowerForth on an Unsupported Workstation.

If you are installing Pinc PowerForth on a workstation that is not currently supported by MPE you should read this section (having read the previous one).

2.3.1 Selecting a Makefile Model.

Generating a Pinc PowerForth system on an unsupported workstation will involve you in the creation of a new makefile specifically for that workstation. We recommend that rather than starting from scratch, you choose one of the makefiles supplied and tailor it to your requirements. E.g. If you implementing Pinc PowerForth on a VAXStation running ULTRIX you would copy the **sun.mk** file a modify the copy of that file to suit the VAX.

The creation of the makefile from an existing one involves three distinct stages. These are outlined below:

2.3.2 Changing the Compiler Directives.

Edit your makefile, correctly naming the C compiler. Most systems call the compiler **cc**. Check how switches are passed to the compiler and how macros are declared from the command line. You are required to be able to specify the following to the compiler:

1) Compilation to object only. This is often specified by a -c switch or a /c switch in the case of MicrosoftC.

2) The name of the object file. This is often specified by a -o switch followed by the name of the object file or /Fo switch in the case of MicrosoftC.

3) The following macro is declared. This is often specified by a -D switch followed by the name of the macro or /D switch in the case of MicrosoftC.

4) The name of the executable file. This is often specified with the -o switch again, but in the case of MicrosoftC it is achieved via the /Fe switch.

5) Link the following objects. This is often specified by simply listing all of the files ending in their object suffixes. E.g. .o for most Unix Compilers and .OBJ for MicrosoftC.

2.3.3 Checking the Make Order.

Some make utilities are order dependent and require items which are prerequisite to the construction of the final goal software to be precede the final goal rule or vica-versa. It therefore advisable to specify exactly which goals must be achieved by a make. This is specified in the following ways on the Sun implementation:

```
all:    fidl pinc cweed boot.dic
```

You may find it necessary to reverse the order of the makefile which you are editing.

2.3.4 Choosing a CModel Implementation.

You should choose a C Model which is appropriate both to the environment in which you wish to run Pinc PowerForth and the size of specific data items in bytes generated by your C compiler.

We recommend that you examine the header file **cmodel.h**. and choose an implementation which is closest to the one you wish.

You can then base your implementation upon this. E.g. If you are implementing Pinc PowerForth on an unsupported Unix workstation you should choose the Sun implementation.

Having chosen an implementation which is close the one you wish, #define an implementation number which is unique. E.g.

```
#define HP9000 9
```

This identifies your implementation. You should then state that you wish to perform this implementation by changing the existing #define IMPLEMENTATION to:

```
#define IMPLEMENTATION HP9000
```

This states that you wish to generate an HP9000 implementation of Pinc PowerForth.

Now **grep** each of the C source and header files and the file **fidl.src** for the string “IMPLEMENTATION ==”. This will produce a list of where all the implementation specific parts of Pinc PowerForth are.

2.3.5 Editing the Source Code for Your Implementation.

Having produced the list of implementation specific areas, you should now use these to edit the source code, header files and **fidl.src** files and produce implementation specific code for your implementation. This code will be encapsulated in the macro processor statements:

```
#if IMPLEMENTATION == y implementation


: : :
: : :
#else
```

We recommend that at first this code be copies of the code of the implementation closest to yours that you chose above. You should change this code incrementally, performing makes at each increment to verify your progress.

2.3.6 Choosing The Cell Size.

You will note that the for each implementation type there is set of declarations concerning the size of CELLs in Pinc PowerForth. This can be found in the header file **cmodel.h**. A Cell is the unit size of data stack items and addresses in Pinc PowerForth. If you are working on a system that supports 32bit integers you would define a CELL_SIZE for your implementation of 4 (four bytes per cell). However, if you are working with a system of where 16bit integers are standard, you would declare a CELL_SIZE of 2. We advise you choose one of the chunks of implementation code regarding Cells and encapsulate it in the appropriate:

```
#if IMPLEMENTATION == y implementation
        : : :
        : : :
#else
```

2.3.7 Choosing The Memory Size.

The choice of Cell Size described above has repercussions upon the amount memory that Pinc PowerForth can address. The size of Ram and the two stacks is defined in the header file **vmsize.h**. You should modify the values according to you requirements.

2.3.8 Changing OS Specifics.

Different operating systems have different file naming conventions. Many of the filenames used by Pinc PowerForth are parameterised and defined in the file **fname.h**. In addition, the file **os.h** contains some of the os specific commands which Pinc PowerForth uses. You should set these accordingly.

2.3.9 Making Pinc PowerForth on a New OS.

Creating a new implementation of Pinc PowerForth is an incremental process. You should run a make after each incremental change to the software. In addition, if the software compiles but fails to perform correctly we recommend that you recompile the software with the **CHECK_STACKS** and **CHECK_INNER** macros defined. Edit the file **misc.h** to achieve this.

Blank Page

Tutorial

3.0 A quick introduction to Forth

The Pinc PowerForth manual does not pretend to be a complete teach yourself Forth book. There are many books on Forth, and in our experience, most programmers buy more than one of them. The bibliography at the end of the manual lists some of these books with our comments about them.

3.1 Writing programs in Forth

Forth is a different sort of computer language. Forth is easy to use because it is interactive, fast because it is compiled, and not as fast as it might be because it is interpreted. Forth is a language with a definite style.

Forth often takes a little longer to learn than other languages. Just like spoken languages, there are many words to learn in Forth before you can use it well. Forth is a language in which very little is hidden from you. Nearly every word that we used along the way to some other function has a name, and is documented in a glossary. As a result of this openness there are many words in the dictionary (type WORDS to see them). Predefined functions in Forth are in fact called words in Forth jargon. These words are stored as a dictionary, and the group of words forming your area of interest - the context in which you work - is known as a vocabulary. For example, words used to define the assembler are kept in a vocabulary called ASSEMBLER. As in all computer languages, there is a jargon to Forth. In this instance the jargon is a technical language, and serves as a set of communication tools so that we can explain our ideas to each other without being bogged down in the minutiae. Persevere, Forth is not only well worth the effort, but is a tool of spectacular productivity in the right hands.

The Forth run-time package is actually a compact combination of interpreter, compiler, and tools. A command or sequence of commands (words) may be executed directly from the keyboard, or loaded from mass storage as if from the keyboard. In this version of Forth, you can also take input from a normal operating system text file, created by a normal (non-Forth editor). Programs in Forth are compiled from combinations of existing words (already in the dictionary), new words as defined by the user, and control structures such as **IF** ... **ELSE** ... **ENDIF** or **DO** ... **LOOP**. New words can be developed interactively using the keyboard/monitor as well using the editor. If you are teaching yourself Forth, get all your books ready in front of the terminal, and try things out as you go along.

The beauty and power of Forth lies in its extendibility and flexibility. New words can be added either at high or low (assembler) level. Forth is one of the very few languages which can define a data structure and how it is used inside a single definition. This ability to create new words known as defining words, which can add new classes of operators to the language, is one of the keys to the extraordinary power of Forth in the hands of an experienced programmer. A bad Forth programmer is just as much a disaster as in any other language.

If your experience of programming has been in traditionally organised languages such as Pascal, BASIC, or C, you will find reading and writing programs in Forth somewhat bizarre at first. Patience brings rich rewards. Forth becomes much easier to understand once you have mastered a few ideas and played with the language. Among the most important aids in using Forth is the choice of word names. Think about the name of a word in advance. Verbs, nouns, and adjectives all have their place in good Forth programming style. Good choice of word names leads to very readable code, as does the use of white space in source Blocks. You can use any character within a word name - the use of printable ones is sensible. Word names can be up to 31 characters long, and all the characters are significant. This version of Forth does not care about the case of characters in Forth word names. "**CAT**" is the same as "**cat**" is the same as "**Cat**". Embedded comments may be as long as you wish without a space or speed penalty in the compiled code.

Forth programs keep most of their working variables on the stack, rather than in named variables, so reading some sections of code can be a little mind-boggling - even for the experienced. The secret is to keep definitions short and simple. Part of making life easy is making sure that you can work out what the code is doing a year from now.

The language lends itself well to bottom-up design and coding. Like the choice of word names, this can be a double-edged sword. There is no substitute for good overall program design, which can only be done properly from the top down. Bottom-up design and coding is excellent, however, for exploring the nuts and bolts of techniques, algorithms, and low level interfaces. The ability to interactively create, test, and produce working code early in the development cycle is invaluable. Early working code also helps to keep your boss off your back, and it enables customers to make sensible reactions and discover specification errors before it is too late. Carefully used this feature can save you a great deal of time.

3.2 Stacks and postfix notation

Forth contains two stacks, one for storing "subroutine" return addresses, and one for storing data. The first stack is called the return stack, and the second is called the data or parameter stack.

The data stack is an efficient method of passing data between the words that make up a Forth program. Any word that needs data takes it from the top of the stack, and puts any results back on top of the stack. The word .S can be used to display the contents of the stack without destroying them. It is a useful debugging tool. Nearly all modern processors provide for the use of stacks, so stack operations are very fast.

The return stack holds the return addresses of all the words that have been called, but have not yet been left. The return stack is also used for storing temporary data that would only get in the way if kept on the data stack. This sort of data includes loop limits and indices, and data taken off the data stack to reduce the amount of stack manipulation that would otherwise occur. There is a set of words used for transferring data between the stacks.

Because stacks are used for data handling, the use of post-fix, or Reverse Polish Notation (RPN), is very suitable. In this form of writing arithmetic expressions,

operands (the data used) come before the operators (how you use the data), for example:

The normal algebraic notation:

at^2 + bt + c

is better expressed for computer evaluation as:

(at + b)t + c

or using normal computer symbols:

(a * t + b) * t + c

which is then expressed in Reverse Polish Notation (RPN) as:

a t * b + t * c +

Notice that the use of brackets becomes unnecessary. This is because of the use of the stack to hold intermediate results. Although the use of a stack is bewildering to begin with, it soon becomes natural, and eventually it is only noticeable on rare occasions. The word "**.**" (pronounced "dot" or "print") is used to print what is on the top of the stack. You can try a few bits of arithmetic. Type in the following words, pressing the "Enter" or "Return" key at the end of each line:

```
1 2 + .
4 5 * .
9 3 / .
1 2 + 3 * .
1 2 3 + * .
1 2 3 * + .
```

Nearly all Forth words remove their data from the top of the stack, and leave the result behind. Words like **+** (add) and ***** (multiply) remove two items, and leave one behind. There is a Forth word **.S** (pronounced "dot-S" or "print-S") which prints out the contents of the stack without destroying the contents. Use it whenever you want to see what is on the stack. E.G.:

```
1 2 3
.s
```

Forth shows 1, 2, and 3 on the stack. Type:

.s

and forth will show them again - they are still on the stack.

So far, we have executed words by typing their names at the keyboard. The next stage is to create new words, but first we must describe how Forth is documented.

3.3 How Forth is documented

Words in Power Forth are documented in a style popular with many Forth programmers. It shows what is on the stack before the word executes (the input), and

what is on the stack after the word has executed (the output). The top of the stack is to the right of the data, and the execution point is marked by two dashes.

The multiply operator, "*****" (an asterisk) takes two parameters on input, and leaves one on output. It is thus shown :-

(n1 n2 — n3)

The parentheses "**(**" and "**)**" are Forth's way of marking a comment. "n1" and "n2" represent two numbers on the stack before the word executes, "n2" being the topmost, and "n3" represents a number left after execution. In the manual Forth words are written in bold capital letters to distinguish them from the lower case letters of the rest of the text. It does not matter which you use in your programs.

Operands are described using the following notation. The notation is described more thoroughly in the section on notation before the glossaries.

OPERAND	**DESCRIPTION**
n1 n2	16 bit signed numbers
d1 d2	32 bit signed numbers
u1 u2	16 bit unsigned numbers
ud1 ud2	32 bit unsigned numbers
addr1	16 bit address
d-addr	32 bit address
b1 b2	bytes
c1 c2	ASCII characters
f	boolean flag
tf	true flag
ff	false flag

3.4 First words in Forth

The only sure way to learn Forth is to use it. Forth programmers spend more time at the keyboard/monitor, because the interactive nature of the language means that words can be tested as soon as they are entered. A result of this feature is that succeeding words, which use previous ones, use tested code. Adherence to the procedure of 'top down design', followed by 'bottom up' coding and interactive testing, leads to very rapid debugging, and successful program generation. Audits of large software projects reveal that over half the time may be spent on debugging. As this is the largest single activity, it is the one to reduce.

To write new words you can either just type them in at the keyboard, or you can use the editor to put source code in a Block file. If you are interested in software management, do read Fred Brook's book, "The Mythical Man-Month".

To use the editor, type EDIT, wait for the editor to load, and select a blank Block to play with. The index command in the editor (usually <escape>-V) lists the top line (a comment line by convention) of every Block in the file. This will allow you see where any blank Blocks are. If you cannot see any blank Blocks just edit the next Block off the end of the file (or start "**USING**" a new Block file). The editor is documented in chapter 4, which includes a description of all its available commands.

If you decide just to enter the examples directly, you do not need to enter the comments preceded by a backslash:

```
\ this is a single line comment
```

or enclosed by parentheses:

```
( this is a
  multi-line
  comment
)
```

Nor do you need to use the same layout. In fact Forth is completely free-form. This means that the position of the words is unimportant, only their order. Power Forth also does not care whether a word is in 'UPPER CASE' or in 'lower case' or in 'MiXeD CasE'.

Forth word names can contain any characters except spaces or nulls (ASCII character 0). It is sensible to use printable characters, but Forth does not actually enforce this.

New Forth words are defined by the word "**:**" (colon) whose first action is to pick up the name that follows and use it to make a new entry in the dictionary. Then, everything that follows up to the next "**;**" (semicolon) defines the action of the word.

By comparison with extended BASICs, "**:**" is equivalent to DEFine PROCedure or DEFine FuNction and "**;**" is equivalent to END PROCedure or END DEFine. For example:

```
: NEW-WORD  ( — )
        CR  ." This is a new word"  CR
;
```

The example word above is called NEW-WORD - when you type:

```
NEW-WORD
```

it will print a new line, print the text:

```
This is a new word
```

and then print another new line. The word **."** (pronounced "dot-quote" or "print-quote") prints out all the characters except the space after **."** up to but not including the next double quotation mark (**"**). The word **CR** is a predefined word that generates a new line - **CR** stands for Carriage-Return.

A new Forth word can contain any words that exist in the dictionary. The new Forth word can be executed by typing its name, or by including it in the definition of another word.

```
: TIMES  ( n1 n2 — )
        * .
;
2 4 TIMES
```

This example will multiply two numbers together and print the result. The word ***** (an asterisk) is Forth's multiply word, and **.** (dot) is the word to print a number. **TIMES** can be used as part of a word that presents information more prettily to the user. First we print a new line using **CR**, then we duplicate the two numbers using **2DUP** and print them out together with the result. Why is the word **SWAP** used? Try it without **SWAP**.

```
( n1 n2 — )
: MULTIPLY
        CR                              \ new line
         2DUP                   \ n1 & n2 for printing
        SWAP                            \ n1 to top of stack
        . ." multiplied by "  . ." equals "
                                        \ print "n1 x n2 = "
        TIMES                   \ calculate and print result
         CR                             \ new line
;

4 3 MULTIPLY
```

We have shown the definition entered on several lines. When you type it in there will be no "**ok**" prompt after the first line, this is to remind you that you have not finished the definition. The Forth system has converted the list of word names into a dictionary entry called **MULTIPLY** and a list of a addresses. These addresses point to the words whose names you entered (ie **CR**, **2DUP**, **SWAP**, **."**, **.**, and **TIMES**). This process is called compilation.

Source text entered from the keyboard is discarded. If you want to keep source code available for re-use, you will need to learn to use the Block editor. This is loaded and run by the word **EDIT**. Use the index command from the editor to see what is in the file. The index command displays the top line of each Block, which, by convention, is a comment line describing the contents of the Block. The next section shows how Forth is documented, and then we show how to create new words.

Suppose we had a section of a program that had to greet people. First, we could define a word to say "hello". We use a dot ("**.**") at the beginning of the name because it is a Forth convention that words which print start with a dot. We use "**:**" (colon) to start a definition (followed by its name) and "**;**" (semicolon) to end it.

```
: .HELLO  ( — )   \ has no effect on the stack
        ." Hello "
;
```

If you now type **.HELLO** <ENTER>, Forth will respond, followed by "ok" to show that there were no errors in the last entry. We now need some words to print out the names of the people we want to greet.

```
( — )
: .FREDA
        ." Freda “
;
: .MARY
        .” Mary “
;
: .NEIL
        .” Neil “
;
: .LINDA
        .” Linda “
;
```

We will also need a word to link these together.

```
: .AND
        ." and “
;
```

We can now define a word to greet all these people. Forth words can occupy as many lines as are needed. You can use line breaks and additional spaces to emphasize the phrasing of the word.

```
( — )
: .GREET
        .HELLO
        .MARY .AND .FREDA .AND .LINDA .AND .NEIL
        CR
;
```

When you type **.GREET** <ENTER>, Forth will respond -

```
Hello Mary and Freda and Linda and Neil
 ok
```

At some stage you will want to see what words are in the dictionary, to do this enter:

WORDS

You will see a long list of words roll past. You can stop the listing by pressing the space bar. Press it again and the listing will continue. Press any other key, and the listing will finish. All the names you see are the names of predefined words in Power Forth, plus any that you have created. All these words are available for you to use, and the predefined ones are documented later in this manual. The words that you write will use these words as their basis.

The secret of writing programs in Forth is to keep everything simple. Remember the KISS method (Keep It Simple, Stupid). Simple things work, and complicated things can be built out of simple things. A programmer’s job is to decide what those simple things should be, and then to design and code them. If the names of the words reflect what they are to do, then the code will be readable and easy to follow. The next sections give an introduction to the components of Forth, and a description of the program control structures available.

3.5 An Introduction To Compiling And Interpreting

The action of compiling involves taking the input text and converting all of it into a machine executable form. The input text is discarded. The executable form is then run. This can produce a program that runs very fast. To change the action of the program, you have to re-type the source text, (or re-edit the file,) then repeat the above process. Although programs run quickly, they take a long while to change. This 'batch' process is used by languages such as C and Pascal.

Interpreting envolves taking the input text, converting some of it into a form the machine can execute, executing it, discarding the executable form, then so on for the rest of the input text. To change the action of the program, you just alter the part of the source text that needs changing, then repeat the above process. This results in programs that are easy to change but slow to run. Early BASIC systems did just this.

Most BASIC systems perform a hybrid of these two, in that they read in all of the text and convert the pre-defined keywords into "tokens"; which is a sort of compilation. Then they interpret the tokens. While this runs faster than pure interpreted code, it is still a lot slower than a compiled code.

Forth also performs a hybrid of compilation and interpretation, but instead of compiling tokens, it compiles the addresses of the words, so that the interpreter has a lot less work to do and consequently is a lot faster. Indeed this method approaches the speed of the more traditional compiler, yet keeps the interactivity (the ability to edit only a part of the program) of the interpreter.

One reason for Forth's speed is that the interpreter is actually very short, only two or three machine instructions on some processors. The relatively slow job of compiling is performed as the text is entered - after you type carriage return. The time taken to do this compilation is very short as far as you are concerned (as it is only one line), but it allows subsequent execution of the code to be very fast.

This interpreter is actually called the Inner Interpreter or Address Interpreter, to differentiate it from the other Forth interpreter, the Outer Interpreter. The outer interpreter is the Forth word that gets your input from the keyboard or from disk, parses out the words and executes them.

So if you enter the words:

1 2 + .

the Outer Interpreter finds and executes the words "**1**"," **2**", "**+**", and "**.**", which do the following:- put a one on the stack, put a two on the stack, add the top two items on the stack leaving one item (the number three), and finally print the top number from the stack.

If the outer interpreter executes a compiler word, such as "**:**", the compiler takes over and instead of executing the words as they are parsed, it compiles them. So that the following sequence:

```
: TEST  ( — )
        1 2 + .
;
```

would cause a word called "**TEST**" to be entered into the dictionary and the words "**1**"," **2**", "**+**", and "**.**", to be compiled as the action of "**TEST**". So that when "**TEST**" itself is executed it will do the same as our previous example. (i.e. add one and two and print three.)

3.6 Defining Words and Immediate words

Remember, all commands to Forth are pre-defined "words" in a "vocabulary" in the "dictionary", consequently Forth can look up the address of a given word for later execution. Some words in Forth change the way the compiler deals with text.

For instance we could define a word that doubles the value given to it.

```
( n1 — n1*2 )
: 2*
        2 *
;
```

The address of the word "**:**" is found and executed by the outer interpreter; the action of "**:**" is to start the compiler (and stop the outer interpreter). The compiler defines a new word whose name comes next ("**2***"), and then compiles into the new word the addresses of the words that follow. This would carry on for ever unless we had a way of stopping it, and this is provided by words such as "**;**" which are "immediate", that is, always executed regardless of what the compiler would otherwise be doing. The action of "**;**" is to stop the compiler and start the interpreter.

There are other types of word (defining words) which are used to create words such as "**:**" - these are one of the keys to advanced use of Forth. To gain a better understanding of compiling/interpreting, defining/immediate words, you should read a good Forth book (see bibliography). All we have attempted to do here is give an introduction. If this seems a little complicated to begin with remember that the basis of Forth is always very simple. Forth is a language built from a number of very simple ideas, rather than one founded on a few complex systems.

3.7 Constants and variables

Many times in a program we need to use a value to represent something, e.g. a type of flower or a bus route number. On other occasions the value corresponds to an actual value rather than an association, the price of roses today, the ASCII code for a special key. Some of this data never changes, it is constant. Other data changes from day to day or minute to minute. These two types of data are represented in Forth by **CONSTANT** and **VARIABLE**. Naming data makes programs easier to write and read, as people remember names more easily than numbers.

Constants are used when the data will not change, or will only be changed when the programmer edits the program (for instance, to change a control key). Constants return their value to the stack.

```
DECIMAL
( — n )
13 CONSTANT ENTER-KEY
1 CONSTANT DAFFODIL
2 CONSTANT TULIP
3 CONSTANT ROSE
4 CONSTANT SNOWDROP
```

In Forth a variable is named and set up by the word **VARIABLE**.

```
( — addr )
VARIABLE FLOWER
```

At this stage a variable called **FLOWER** has been declared with an initial value of 0. When you use **FLOWER** the address of the data is given. Forth uses **@** (“fetch”) to fetch the data from the address, and **!** (“store”) to store data into an address. When the variable was created it was given the value 0. If another value is needed we can change it.

```
DAFFODIL FLOWER !
```

Later on in a program we can use the value of **FLOWER** to change the way the program acts. Part of the program might be about to draw the flower, and we need to set the colour properly.

```
( — )
: DAFF
  FLOWER @ DAFFODIL =
  IF  YELLOW INK  ENDIF  ;
```

To make a program wait until the ENTER key is pressed, you can try the code below.

```
( — )
: WAIT-ENTER
  BEGIN  KEY ENTER-KEY =  UNTIL  ;
```

In most words data is passed from word to word using the stack. If you find the stack usage getting too complex, try splitting the word into other words which only use one or two items on the stack. On other occasions you will find that all the words need to refer to the same value which controls what happens on this run of the program (say, the type of flower). In these cases the use of a variable is appropriate. As your Forth skills improve, you will find that you use fewer variables.

3.8 Control structures

The flow of control in a program is nominally sequential; a control structure is a construct that allows you to alter the control flow to include a branch or a loop. This change is often dependant upon the value of some piece of (runtime) data. This gives a program the power of choosing to do this if one thing happens, or that if another thing happens. Control structures in Forth allow execution and looping determined by values on the stack. Although the user is not necessarily aware of it, control structures are usually implemented by means of words that execute at compile time (immediate words), and compile other words that actually execute at run time. Techniques like these allow error checking to be implemented as well. Control structures must be used

inside a colon definition; they cannot be directly executed from the keyboard. Any one structure must be written entirely within one definition; you cannot put the **IF** in one word and the **ENDIF** in another. Control structures can be nested inside one another, but they must not overlap.

3.8.1 **IF** ... **ENDIF**

flag IF <true words> ENDIF

The flag on the top of the stack controls execution. If the flag is non-zero (true), the words between **IF** and **ENDIF** are executed, otherwise they are not.

Like most Forth words **IF** consumes the flag used as input to it. If the value of the flag must be used again it can be duplicated by **DUP.** In the case of **IF** ... **ENDIF**, where the value may only be needed between **IF** and **ENDIF**, the use of **?DUP**, which only duplicates a number if it is non-zero, may be more appropriate.

```
: TEST  ( f — )
        IF  ." top of stack was non-zero “  ENDIF
;
1 TEST top of stack was non-zero  ok
0 TEST  ok
```

3.8.2 **IF** ... **ELSE** ... **ENDIF**

flag IF <true words> ELSE <false words> ENDIF

This structure behaves just like **IF** ... **ENDIF** above except that an alternate set of words will execute when the flag is false (zero). E.G.:

```
: TEST  ( w — )
        IF
                ." top of stack was non-zero “
         ELSE
                .” top of stack was zero “
        ENDIF
;
1 TEST top of stack was non-zero  ok
0 TEST top of stack was zero  ok
```

3.8.3 **DO** ... **LOOP** and **DO** ... n **+LOOP**

limit index DO <loop words> LOOP
limit index DO <loop words> increment +LOOP

This structure is similar to BASIC’s:

```
FOR X = n1 TO n2
        ...
NEXT
```

To use this structure, place the limit value and the starting value of the loop index on the stack. **DO** will consume this data and transfer it to the return stack for use during execution of the loop. **LOOP** will add one to the index and compare it to the limit. If the index is still less than the limit the loop will be executed again. **+LOOP** behaves similarly except that the increment on the stack is added to the index each time round the loop.

You can get out of the loop early with the words **LEAVE** or **?LEAVE**.

LEAVE cleans up the return stack, and execution then resumes after the **LOOP** or **+LOOP**. Note that words between **LEAVE** and **LOOP** or **+LOOP** are not executed.

?LEAVE behaves like **LEAVE** except that the loop is left only if the top of the stack is non-zero. This is a useful word when checking for errors.

DO ... **LOOP** structures may be nested to any level up to the capacity of the return stack. The index of the current loop is inspected using "**I**" . The index of the next outer loop is inspected using "**J**".

WATCH IT -

If you use the return stack for temporary storage after **DO**, you must remove it before **LOOP** or **+LOOP**.

If there is data on the return stack **I** and **J** will return incorrect values..

The loop will always be executed at least once. If you do not want this to happen you must add extra code around the loop or use **?DO**, which only executes if the limit is not the same as the index.

Example:

```
: TEST  ( — )
        10 1 DO  I .  LOOP
;
TEST 1 2 3 4 5 6 7 8 9  ok
: TEST2  ( — )
        10 1 DO  I .   3 +LOOP
;
TEST2 1 4 7  ok
```

3.8.4 **?DO** ... **LOOP** and **?DO** ... n **+LOOP**

limit index DO <loop words> LOOP
limit index DO <loop words> increment +LOOP

These structures behave in the same way as **DO** ... **LOOP** and **DO** ... n **+LOOP** except the loop is not executed at all if the index and the limit are equal on entry. For example if the word **SPACES** which prints n spaces is defined as:

```
: SPACES      \ n —
        0 DO  SPACE  LOOP
;
```

the phrase **0 SPACES** causes 65536 spaces to be displayed, whereas the definition below displays no spaces.

```
: SPACES      \ n —
        0 ?DO  SPACE  LOOP
;
```

3.8.5 BEGIN ... AGAIN

BEGIN <words> **AGAIN**

This structure forms a loop that only finishes if an error condition occurs, or a word such as **ABORT** or **QUIT** is executed. The first example will read and echo characters from the keyboard forever, the second will exit when the ENTER key is pressed. Example:

```
DECIMAL
: TEST
        BEGIN  KEY EMIT  AGAIN
;
: TEST2
        BEGIN  KEY DUP 13 =
                IF ABORT ENDIF
                EMIT
        AGAIN
;
```

3.8.6 BEGIN ... UNTIL

BEGIN <words> flag **UNTIL**

This structure forms a loop which is always executed at least once, and exits when the word **UNTIL** is executed and the flag (on the data stack) is true (non-zero). If you need to use the terminating condition after the loop has finished use **?DUP** to duplicate the top item of the stack if it is non-zero.

BEGIN ... **UNTIL** loops may be nested to any level. Example:

```
: TEST
        BEGIN  KEY DUP EMIT 13 =  UNTIL
;
```

3.8.7 BEGIN ... WHILE ... REPEAT

BEGIN <test words> flag **WHILE** <more words> **REPEAT**

This is the most powerful and perhaps the most elegant (though certainly not some purists choice) of the Forth control structures. The loop starts at **BEGIN** and all the words are executed as far as **WHILE.** If the flag on the data stack is non-zero the words between **WHILE** and **REPEAT** are executed, and the cycle repeats again with the words after **BEGIN**.

This structure allows for extremely flexible loops, and perhaps because it is somewhat different from the structures of BASIC or PASCAL, this structure is often somewhat neglected. It does however, repay examination. Like all the structures it can be nested to any depth, limited only by stack depth considerations. In the example below, the console is polled until a key is pressed, and a counter is incremented while waiting. Example:

```
VARIABLE COUNTER
: TEST
        0 COUNTER !
         BEGIN
                ?TERMINAL 0=
        WHILE
                1 COUNTER +!
        REPEAT
;
```

3.8.8 CASE ... OF ... ENDOF ... ENDCASE

CASE key
value1 **OF** <words> ENDOF
value2 **OF** <words> ENDOF
...
<default words> (otherwise clause)
ENDCASE

The **CASE** statement in Power Forth is the result of a competition for the best **CASE** statement. The competition was run by Forth Dimensions, the journal of the Forth Interest Group (FIG) in the USA. A large number of **CASE** statements were proposed, but this one has stood the test of time as it is secure, easy to use and understand, and easy to read. It was invented by Dr. Charles E. Eaker, and first published in Forth Dimensions, Vol. II number 3, page 37.

CASE statements exist to replace a large chain of nested **IF**s, **ELSE**s, and **ENDIF**s. Such chains are unwieldy to write, prone to error, and lead to severe brain-strain.

The function of a **CASE** statement is to perform one action dependent on the value of the key passed to it. If none of the conditions is met, a default action (the otherwise clause) should be available. Note that a value to select against must be available before each **OF** against which the entered parameter may be tested. The select value is top of stack, the parameter is next on stack (by requirement); **OF** then compares the two values, and if they are equal, the words between **OF** and **ENDOF** are executed, and the program continues immediately after **ENDCASE**. If the test fails, the code between **OF** and **ENDOF** is skipped, so that the select value before the next **OF** may be tested. If all the tests fail the parameter is still on the data stack for the default

action, and is then consumed by **ENDCASE**. Additional control structures can be used inside **OF** .... **ENDOF** clauses. Example:

```
: STYLE?      ( n — )
        CASE
                1  OF  ." Mummy, I like you"          ENDOF
                2  OF  ." Pleased to meet you"        ENDOF
                3  OF  ." Hi!"                              ENDOF
                4  OF  ." Hello"                            ENDOF
                5  OF  ." Where's the coffee"         ENDOF
                6  OF  ." Yes?"                             ENDOF
                          ." And who are you?"
        ENDCASE
;
```

The phrase

n STYLE?

will select an opening phrase according to the value of n. If n is in the range 1..6 a predefined string is output, for any other number the default phrase "And who are you?" is output. Case statements are often used to select actions based on ASCII characters. In an editor sections of code like the one below are often found.

```
CASE
        KEY
        ASCII I  OF  INDEX                  ENDOF
        ASCII M  OF  MOVE-SCREENS    ENDOF
        ASCII D  OF  DIRECTORY          ENDOF
        ..................
ENDCASE
```

In order to extend the usefulness of the basic **CASE** structure we have added three extensions to the standard Eaker **CASE**.

The first **?OF** allows the use of a logical test rather than equality. If you need to test whether or not a character is in the right range the following replaces a large number of **OF** ... **ENDOF** sets. The word **WITHIN?** returns a true value if the value is between (or equal to) the lower and upper limits. **?OF** consumes the flag given to it.

```
DUP  32 127 WITHIN? ?OF ...  ENDOF
```

The second extension **END-CASE** behaves just like **ENDCASE**, except that it does not **DROP** anything from the stack, so allowing a default clause to consume the select value without having to **DUP**licate it.

The third extension **NEXT-CASE** compiles a branch back to the **CASE**, so producing a loop that exits via one the **OF** ... **ENDOF** or **?OF** ... **ENDOF** phrases. Such a loop performs a different exit action for each condition. The intention of this structure is to allow a formal method of constructing loops with more than one exit and exit action. Such loops are often necessary when dealing with text entry. **NEXT-CASE** consumes no data. Example:

```
CASE  KEY
        13      OF      <carriage return action>        ENDOF
        10      OF      <linefeed action>               ENDOF
        DUP 32 127 WITHIN?
                ?OF     <normal action>                 ENDOF
                CR ." Character code “ .  .” is invalid" CR
NEXT-CASE
```

3.9 Text and Strings

A string is a sequence of characters. A Forth string is stored as a count byte followed by that many characters. There are not many words in a standard Forth that deal with strings, but they do allow the user to build words that will perform any required function. To put a string into a word use **“”** (double-quote double-quote) which compiles a string into the dictionary, and returns its address when the word executes.

```
( — addr )
: HELLO$
        “” Hello there “
;
```

The string will start with the “H” and end with the space before the quotation mark. The address returned by **HELLO$** points to the count byte. To convert this address to the address of the first character and the number of characters, the word **COUNT** is used.

When a string is to be printed, it is usually done by the word **TYPE**, which needs the address of the text to be printed and the number of characters to be printed. To print the string above we would use:

```
HELLO$ COUNT TYPE
```

To pick individual characters out of a string you simply add the number of the required character to the start address and fetch it. For the first ‘l’:

```
HELLO$ 3 + C@ EMIT
```

To print the sequence ‘lo t’:

```
HELLO$ 4 + 4 TYPE
```

Using this type of string extraction you can generate words which perform the equivalent of the usual BASIC string handling words. The principal word to fetch strings from the keyboard is **EXPECT** which requires the address of a buffer in which to put the string, and the maximum number of characters to read. To create an 80 byte buffer, read a string into it, and inspect the contents of the buffer, we create a buffer called **BUFFER$** and then reserve another 78 bytes (the variable reserves two), read 80 bytes into it, and then display the contents of **BUFFER$**.

```
CREATE BUFFER$  80 ALLOT
BUFFER$ 80 EXPECT
( now enter a string at the keyboard )
BUFFER$ 80 DUMP
```

You will see that **BUFFER$** contains the text you entered without a count byte. Inside Forth there is an area called the terminal input buffer (its address is returned by the word **TIB**). The word **QUERY** reads a line of text into **TIB**. The word **WORD** then extracts a sub-string bounded by a specified character, and copies the string to the end of the dictionary as a counted string (count byte + characters), returning the address at which it left the string. Try entering the word below:

```
: T     \ just another test word
        BL WORD 40 DUMP
;
T xxxx yyyy zzzz
```

3.10 Print Formatting

Forth has a very powerful number string formatting system. It is quite different from that supplied with BASICs. To print a number as pounds and pence (or dollars and cents, or francs and centimes), try the following:

```
HEX
: .POUNDS
        060 EMIT
        S>D  <# # # ASCII . HOLD #S #>  TYPE
;
DECIMAL
5050 .POUNDS
```

The phrase **060 EMIT** prints the pound sign. Number conversion is started by **<#** and finished by **#>**. The word **<#** needs a double number (**S->D** converts single numbers to double). Numeric conversion then proceeds LEAST SIGNIFICANT DIGIT first. For each **#** a digit is converted. The word **HOLD** takes a character and inserts it into the character string being generated. So the phrase

```
# # ASCII . HOLD
```

produces the two least significant digits and a decimal point (the pence portion). The word **#S** converts the rest of the number, producing at least one digit. Numeric conversion is finished by **#>** which leaves the address of the generated string, and the number of bytes in the string. **TYPE** then prints the string.

You can easily produce your own number conversion formats. Suppose we wanted to print numbers as pounds and pence, with six figures before the decimal point, two after it, and leading zeros suppressed except in the character before the decimal point. The format we want is ‘’xxxxy.yy’ where x is a digit or a blank and y is a digit.

We need to generate a word **#B** which produces a digit if possible or a blank, if number conversion has finished.

In between **<#** and **#>** the number being converted is in double number form. When a digit is converted by **#** it is divided by the current base, the remainder is converted to a character to be output, and the quotient is returned. Thus the function of **#B** is to allow numeric conversion if the number is non-zero, otherwise to insert a blank into the output.

```
( d — t/f )
: D0=
        OR 0=
;
( d1 — d2 )
: #B
         2DUP D0=
        IF      BL HOLD
        ELSE   #
        ENDIF
;
```

The word **BL** returns the character code for a space. Now we can generate **.P** which will print a double number as eight characters with a decimal point.

```
( d1 — )
: .P
        <#  # #  ASCII . HOLD  #  #B #B #B #B  #>
        TYPE
;
500,00 .P
```

When you enter a number that includes a comma, it is treated as double number. The position of the comma has no significance, except that the number of digits after it is held in the variable **DPL**. Try:

```
5,0000 .P  DPL @ .
500,00 .P  DPL @ .
```

Both numbers are converted to the same value, but the system variable **DPL** tells you how many digits were entered after the comma. If no comma had been entered, the number would have been treated as a single number, and the value of **DPL** would have been -1.

If you want to explore print formatting more fully, and gain yourself fame for the more exotic style of programming, try formatting time and date from a 32 bit number of seconds. Changing the number conversion base in the middle of print formatting words can be very useful.

3.11 Vocabularies

In constructing an application written in Forth a large number of new words are generated. With the already large number provided in the nucleus, problems can arise in simply remembering what the words mean. Other problems can arise because in one part of an application, **ATTACH** may refer to attaching an interrupt to a piece of code (as it does in the interrupt code), and in another section, it may refer to an operation on a file, and again may refer elsewhere to the application performed by the robot under control.

A method is needed to control the environment in which we are working. In the examples above the word **ATTACH** has different meanings in different contexts. The context in which a word exists is its vocabulary. Vocabularies are created by the phrase:

VOCABULARY <vocabulary-name>

For example the vocabulary of words dealing with the robot might well be called **ROBOTICS**, and would be defined by:

VOCABULARY ROBOTICS

The Forth system is told which vocabulary words are defined into by the word **DEFINITIONS**, which sets the vocabulary that words are currently to be defined in. After the phrase:

ROBOTICS DEFINITIONS

all new words will be part of the **ROBOTICS** vocabulary, and you could list all the words in that vocabulary by typing:

ROBOTICS WORDS

Having defined which vocabulary new words are built into, we must now define which contexts are relevant when searching for word names. For example, moving a robot arm might need access to floating point words in vocabulary **F-PACK**, the graphics words in **GRAPHICS** for console displays, and on-line manual words in **MANUAL**. To cope with all this we also need a way to start at the beginning again.

The word that means 'search the minimum' is **ONLY** which sets Power Forth to search only a tiny vocabulary called **ROOT**. Most of the common words are in **FORTH** which is the main vocabulary. Thus the phrase:

ONLY FORTH

resets the system to only use **FORTH** (and the little **ROOT**). We can add another vocabulary to be searched with the word **ALSO** which adds the new vocabulary so that its searched first. After executing:

ALSO F-PACK

the **F-PACK** vocabulary will be searched first, then **FORTH**, and finally **ROOT**. To provide the complex order described earlier, the following phrase can be used:

```
ONLY FORTH
ALSO F-PACK  ALSO GRAPHICS  ALSO MANUAL
ALSO ROBOTICS DEFINITIONS
```

It is usual for the vocabulary into which words are defined to be the first in the search order, and so the last one specified. There are two reasons for this.

Firstly, the Forth-83 specification states that "**:**" (colon) used to start a high level definition, makes the defining vocabulary the one that is searched first. The consequence of this is that if we had specified:

```
ONLY FORTH
ALSO ROBOTICS DEFINITIONS
ALSO F-PACK  ALSO GRAPHICS  ALSO MANUAL
```

the initial search order would have been:

MANUAL GRAPHICS F-PACK ROBOTICS FORTH ROOT

but after the first colon, the order would be:

ROBOTICS GRAPHICS F-PACK ROBOTICS FORTH ROOT

If you must define fancy search orders that need the defining vocabulary searched late, you will have to define immediate words so that the search order can be changed inside a colon definition.

Secondly, when the **ROBOTICS** vocabulary is being defined into, it is most likely that other words in the same **ROBOTICS** context will be required, and if duplicate names exist, it is the ones in the **ROBOTICS** context that are most likely to be needed.

You can get a list of all the vocabularies that have been defined by typing:

VOCS

and you can see the search order used by typing:

ORDER

The following Forth words are involved in vocabulary control, and they are all documented in the main glossary:

```
CONTEXT CURRENT                 \ pointers
FORTH ROOT                      \ vocabularies
VOCS ORDER WORDS                \ display words
ONLY ALSO DEFINITIONS PREVIOUS SEAL UNSEAL
```

Blank Page

Source Code And Editing

4.0 Introduction

Pinc PowerForth can compile source code from any standard ASCII file generated by a standard program/text editor. Word processors that leave control codes or use the top bit for control purposes should be avoided, since eight-bit characters are used (i.e. not seven-bit) to allow for Continental European characters.

4.0.1 Configuration

Within Pinc PowerForth the editor to be used and the current text file can be defined like so:

```
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.

4.0.2 Compiling from Text files

From within Pinc PowerForth, a text file can be loaded by using either of these two phrases:

```
ALL FROM-FILE <path-name>          \ include named file
ALL FROM                                \ include current file
```

Thus to load from the file DRIVERS.FTH use the phrase:

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

4.0.3 Examples Of Text File Usage

Here follows examples of text file usage, showing the text typed in, the computers responce, and a commentary:

To set the default editor to a program called “ ” type:

EDITOR-IS VI

To check this, type:

.ED

Forth responds with

Current editor: vi ok

To set the default file to “ . ”, type:

USE MAIN.FTH

To check this, type:

.FTH

Forth prints:

Default source file: MAIN.FTH ok

To edit the file using the “ ” editor, type:

ED

Edit “ . ” using the “ ” editor, then exit the editor.

To load the file, type:

ALL FROM

The file “ . ” is loaded.

4.0.4 Text File words Glossary

Listed next 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.

.ED —
“dot-ed” or “print-editor”

Print the path & name of the current text editor.

.FTH —
“dot-fth” or “print-fth”

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

;P —
“semi-P”

End Page. (For file compatibility with the Modular Forth paged text file loader) Causes **FROM** and **FROM-FILE** to stop using the current page. Has a similar effect as **;S** in screen files.

(—
“open-paren” or “paren”

Comment. The comment **(** may be freely used while interpreting or compiling. The number of characters in the comment may be from zero to the number of characters remaining in the input stream up to the closing parenthesis. The parenthesis comment works over multiple lines. This version of the **(** comment differs from the normal Forth **(** comment because its end is marked by a white-space-delimited closing parenthesis. E.G.

```
...
2DUP  (
        save adr & # for later
        NB: see modification notes
        )
 INIT
...
```

ED <text-file-name> ; —
“ed”

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

EDITOR-IS <editor-file-name> ; —
“editor-is”
Set the file-name of the editor you wish to use.

FROM n1 n2 —
“from”
Get text input from the current file. Files can be nested to any depth; that is files can include input from other files.

FROM-FILE <text-file-name> ; —
“from-file”
Get text input from the file <text-file-name> typed after **FROM-FILE**. Files can be nested; that is files can include input from other files.

USE <text-file-name> ; —
“use”
Set the default text file you wish to **USE**.

The Operating System

5.0 Operating System and File Interface

Forth users are split into two camps on the subject of disc usage.

> There are those who insist on writing their own device drivers, and use no file structure at all. The reasoning behind this approach is to extract the maximum performance from the system. Such a technique is particularly appropriate for small dedicated machines where performance must be at its highest on nearly inadequate hardware.
>
> The other camp performs all input and output through calls to a host operating system. The reason for implementing Forth this way is ease of implementation and, more importantly, software portability. The operating systems implementors have gone to some effort (mostly) to make their systems efficient and reliable, so why re-invent the wheel?

Pinc PowerForth is firmly in the second camp. We want you to be able to use all the devices the operating system makes available to you, whether we know about them or not. You can copy Pinc PowerForth to hard discs, or in fact to any mass storage device fitted to your computer. If you are using an operating system with device independent i/o (input/output) you can take full advantage of it. We get another advantage as software vendors in that we can cope with any disc format produced by the hardware vendors, the hardware vendor did all the work when writing the device driver for the operating system.

Pinc PowerForth supports the stream interface method used by C The wordsets provided are the same for whatever Operating System your are running Pinc PowerForth on. We will therefore refer to your operating system by the generic acronym OS throughout the rest of this chapter.

5.1 File handles

In order to use a file (called a path when used with its directory as well) your OS is given the full path name of the file so that the file can be opened, and then the OS returns a magic number called the file handle which is used from then on to refer to the file.

This approach only requires the user to store one number in order to refer to the file, and it is additionally sanctified because this is the approach used by UNIX.

This restriction forces us to save the path name if we need it again after opening the path. To do this Pinc PowerForth allows the use of PCBs, or Path Control Blocks. A path control block consists of a cell sized handle (-1 indicates not opened), followed by a name field. Consequently the words to open and create files (paths) have two forms, one for taking the file name from a string, the other for taking it from a PCB.

Another, and perhaps the major benefit of using the handle approach, is that with it, an OS implements device independent i/o. This means that all devices and files are treated in the same consistent way. Thus **EMIT** can easily be directed to write to a file.

5.2 Simple file handling

In these examples both the direct handle and the PCB approach will be used.

5.2.1 Using path control blocks

First of all we must create a path control block for the file handle and name data.

```
PCB TEST-FILE
```

The word **PCB** creates the path control block. Then the path name must be inserted into the path control block:

```
TEST-FILE PATHNAME TEST.COM
```

And before reading from, or writing to, the file TEST.COM, you must open it:

```
TEST-FILE OPEN-PATH-PCB .
```

The word **OPEN-PATH-PCB** returns an error code (0=success, other=failure code). Inside a colon definition, you must use a different tactic, especially if the file name used by **TEST-FILE** is likely to change, say if the same operation is to be performed on a number of files. In this case, you can use the word **SET-PATHNAME** to copy a path name from a counted string to a PCB. For example:

```
: USE-F1
        “” TEST1.COM" TEST-FILE SET-PATHNAME
;

: OPEN-TEST
        TEST-FILE OPEN-PATH-PCB ?DUP
        IF      ." Error “ DECIMAL .
                .” on opening file “ TEST-FILE .PCB
                CR
                 ABORT
         ENDIF
;
```

USE-F1 will copy the file name TEST1.COM into path control block **TEST-FILE**. The file can now be opened by **OPEN-TEST**. The data in the file can now be read.

Sequential operation is very straight-forward, each operation reads or writes data immediately after the previous operation. If you need to hop around the file, you need to use random access, in which you specify the record number you want, where the record length is as discussed above. To use random access let us assume a record size of 64 bytes. We can now write a word to position the file to record n and then read the record to a buffer at the given address.

```
DECIMAL
( address n )
: READ-TEST
        64 UM*  TEST-FILE HANDLE SEEK-PATH
        ABORT" Can’t seek that record"
        64 TEST-FILE HANDLE READ-PATH
        ABORT" Can’t read that record"
        DROP
;
```

When you have finished with the file, it must be closed. Under your OS, data written to files is buffered, and so issuing a write command to OS does not guarantee that the data is actually written to disc. This can only be enforced by actually closing the file.

```
TEST-FILE HANDLE CLOSE-PATH DROP
```

5.2.2 Using pure handles

To open the file is very simple:

```
VARIABLE TEST

: OPEN-TEST ( — )
        “” TEST.COM" OPEN-PATH
         ABORT" Can’t open file"
        TEST !
;
OPEN-TEST
```

Because we are prepared to dispose of the file name we do not have to establish the structure in which to preserve it. What we do have to do is to preserve the handle throughout the life of the open file. This is done by the variable **TEST**.

The function that reads the random records is very similar to the previous one.

```
DECIMAL
( address n — )
: READ-TEST
        64 UM*  TEST HANDLE SEEK-PATH
        ABORT" Can’t seek that record"
        64 TEST HANDLE READ-PATH
        ABORT" Can’t read that record"
        DROP
;
```

And the file is finally closed in the same way as before.

```
TEST HANDLE CLOSE-PATH DROP
```

5.3 Using Files For Text Input/Output

The Forth words **EMIT KEY KEY? CR TYPE** and **EXPECT** are deferred. This means that the address of a Forth word actually executed is held and executed by the deferred word. CR is normally defined in terms of **EMIT**, but can be altered for specific requirements. The ability to change the device used for a function is known as ‘re-directable i/o’ and is a valuable feature of modern systems.

If you write a word to perform byte by byte i/o to another device, assign that word to perform the action of **EMIT** or **KEY**, you will have changed the device used for that function. This idea can be extended to using operating system files for text input and output.

Under Pinc PowerForth it is not necessary to re-assign **EMIT** for output, as it uses an internal variable **OP-HANDLE** to contain the handle of the path used for output. You can store and access handles into this varaible using the words **OP-HANDLE@** and **OP-HANDLE!**. If the handle of a file is stored into **OP-HANDLE**, output will be to that file instead of the screen.

5.3.1 OS interface glossary

This glossary details the words used to interface to OS and the words needed in turn to use these functions.

.PCB pcb-addr —
"dot-p-c-b"

Prints out the path name in the given path control block

BYE —
"bye"

If Pinc PowerForth is running stand-alone this word will close all open files, via an exit(0) and return to the OS or program that invoked it. If Pinc PowerForth is intergrated into an application and has been called as a standard C function, **BYE** will return control to the part of the application that called Forth.

CLOSE-PATH handle — status-code
"close-path"

Close a file, updating the directory if necessary. Note in particular that under some operating system implementations, data is buffered, and may only be completely written to the disc when the file is closed.

CONSOLE —
"console"

Sets the words **EMIT** and **TYPE** to use the system keyboard and display. Useful after redirecting output to (say) a printer, and the default output is needed again. Often used in the form:

PRINTER <words> CONSOLE

so that the output of the words between printer and console are sent to the standard printer.

COOKED handle —
"cooked"

Input and output to the path is processed by the operating system. This function is meaningless for files on some operating systems but is essential on systems such as MSDOS where <CR><LF> pairs are converted into <LF> during reading and vice-versa during writing. See RAW for a fuller explanation.

CREATE-PATH string — handle 0 | error-status
"create-path"

Given the address of a string containing a path name, the path create OS call is made. On success, the handle and 0 are returned, otherwise the error status is returned.

CREATE-PATH-PCB addr — status
"create-path-p-c-b"

Given the address of a PCB, the path is opened, and the OS status is returned.

EMIT b —
“emit”
The ASCII character in the low byte is displayed. **EMIT** is a deferred word, and the default action is to send the character to the output whose handle is held in the **OP-HANDLE** internal variable.

HANDLE pcb-addr — handle
“handle”
Returns the handle allocated to a path control block. Unopened PCBs will return a handle of -1.

IP-HANDLE@ — handle U
“i-p-handle fetch”
Returns the handle of the current input stream. path. Used by **EXPECT**, **KEY** and **KEY?**

IP-HANDLE! handle — U
“i-p-handle store”
Makes the given handle the current input stream. path. Used by **EXPECT**, **KEY** and **KEY?**

OP-HANDLE@ — handled U
“o-p-handle fetch”
Returns the handle of the current output stream. Used by **EMIT** and **TYPE** for output.

OP-HANDLE! handle — U
“o-p-handle”
Makes the given handle the current output stream. Used by **EMIT** and **TYPE** for output.

OPEN-PATH addr-z — handle 0 | error-code
“open-path”
Find and open the path named in the indicated ASCIIZ string, and return either the handle & 0 if successful, or the OS error code if unsuccessful.

OPEN-PATH-PCB pcb-addr — status
“open-path-p-c-b”
Given the address of a PCB, the PCB’s stored pathname is opened and the OS error code returned.

P-NAME pcb-addr — name-address
“p-name”
Given a path control block, the address of the path name is returned.

PATHNAME pcb-address —
“path-name”
Used in the form:

<pcb-address> PATHNAME <file-spec>

Format a OS path specification into the path control block.

PCB — (defining)
“p-c-b” — pcb-address (child)

Used in the form:

PCB pcb-name

A path control block is named and space for it allocated. When pcb-name is executed, the address of the PCB is returned. The PCB consists of a cell sized area containing the file handle, plus a field for the path name.

PREV-PCB — fcb-addr
“prev-pcb”

A file control block used to store the name of the previous file used as the screen file.

RAW handle —
“raw”

Turns off character processing by OS of file input and output. The OS standard input and output are set to **COOKED** by Pinc PowerForth on entry, and are returned to **COOKED** on exit.

READ-PATH addr len handle — # 0 | error-No error-No
“read-path”

Attempts to read ‘len’ bytes from the path whose ‘handle’ is given into a buffer at ‘addr’. The actual number of bytes read, and the OS status code are returned. If an error occurs (status non-zero) two error-codes are returned.

SAVE —
“save”

use in the form:

SAVE <path-name>

Saves an executable image of Pinc PowerForth’s dictionary up to **HERE** (i.e. all the words added since the last initialisation or **COLD**) to the specified file. The image may then be executed from the operating system prompt by typing its filename as the first argument to Pinc PowerForth. The saved file is a complete executable image. Any start-up vectors modified will be preserved so that turnkey operation is possible.

SEEK-PATH d handle — status
“seek-path”

The current position within the file whose handle is given is set to d. The value d is a double (32-bit) number representing a position d bytes from the start of the file. This is the function that allows random access in a file.

SET-PATHNAME $addr pcb-addr —
“set-path-name”

Parse a file specification from a Forth counted string, into the path control block. The whole path control block is initialised, and any previous information in it will be destroyed. Do make sure that any path previously used by this PCB has been properly closed. The string address is the address of the string’s count byte. See **PATHNAME**

SYSTEM —
“System”

use in the form:

“” <command> [<parameters>]" System

This is an implementation of the OS EXEC function. The file <command> is loaded and executed and the parameter string <parameters> is passed to it. The program <command> is loaded outside the memory area used by Forth, and exit from <command> is back to Forth.

TYPE addr n —
“type”

The len characters at address addr are sent to the path whose handle is in the internal variable **OP-HANDLE**. Nothing is displayed if n is zero. See **EMIT OP-HANDLE KEY KEY? IP-HANDLE**

WRITE-PATH addr len handle — count good-status | error-No
error-No
“write-path”

Writes ‘len’ bytes from a buffer at address ‘addr’ to the path whose ‘handle’ is given. The ‘count’ returned is the number of bytes actually transmitted. If the ‘count’ equals the ‘len’ and ‘good-status’ (zero) are returned, the write has suceeded. If the status is good but the ‘count’ and ‘len’ are not equal then the disk is full. Otherwise the operation failed.

Vectored Execution

6.0 Introduction

It is often of value to be able to change the action of a word without re-compiling. Examples are: (1) changing an i/o word from one device to another without changing the high level code; and (2) allowing the factoring of code to be less dependant on compilation order, because an early-compiled word can use a later-compiled word.

6.1 Vectored Execution

This ability is known as Vectored or Defered execution, and is provided in Pinc PowerForth. There are two stages envolved: (1) declaring a word whose action can be altered; and (2) altering the action.

6.1.1 Declaring A Vectored Word

The defining word to declare a vectored or defered word is called **DEFER**. As an example of its use, the definition of **KEY** in the Forth nucleus is as follows:

```
DEFER KEY  ( — char )
```

At the moment the action of **KEY**, if executed, would be a word called **CRASH** which causes an error abort. The desired action(s) of **KEY** will be asigned later.

To demonstrate how **DEFER** works we can define it using **DOES>** (whose run-time action is to return the address of a defined word's parameter field), like so:

```
: DEFER
        CREATE  [‘] CRASH ,        \ create word with default action
         DOES>  @ EXECUTE          \ execute current action
;
```

Words defined by **DEFER** (e.g. **KEY**) will execute the word whose execution address has been placed in their parameter field. Pinc PowerForth actually uses a much faster machine code routine for **DEFER** but the example above demonstrates the idea properly.

6.1.2 Assigning An Action

The words to assign an action to a defered word are **ASSIGN** and **TO-DO**. Continuing the **KEY** example given above, a separate word **(KEY)** is defined which actually does the work, by the phrase:

```
ASSIGN (KEY) TO-DO KEY
```

As an other example, suppose a CAD application program uses a number of different devices in a common way, and needs to switch between them at run-time:

```
\ define device drivers

: PLOTTER-O/P   .... ;
: TAPE-O/P      ....  ;

DEFER WRITE                 \ o/p driver

\ output redirection

: DEVICE  ( — )       \ select device
        ?DEVICE             \ get device id
         CASE
                PLOT OF  ASSIGN PLOTTER-O/P TO-DO WRITE ENDOF
                TAPE OF  ASSIGN TAPE-O/P            TO-DO WRITE ENDOF
        ENDCASE
;

\ use like this

DEVICE WRITE                \ select device and write
```

6.2 Supplied deferred words

The following words in Pinc PowerForth are deferred, and are documented in the main glossary:

NORMAL-DRIVERS
KEY? KEY
EMIT TYPE
CR PAGE CLS
GOTOXY GETXY
INV-ON INV-OFF
HALF-ON HALF-OFF
UNDER-ON UNDER-OFF
CURSOR-ON CURSOR-OFF
FMT-DATE FMT-TIME
PROMPT
EXPECT
QUIT ABORT ERROR
LOADER
USER-INIT
$CREATE
FIND
WORD
I-LOOP C-LOOP
NUMBER?

6.2.1 Glossary of vectoring words

(TO-DO) cfa1 —
“paren-to-do”

The run-time code compiled by **TO-DO**. The parameter pfa1 is converted to a cfa and stored at pfa2, which is assumed to be the parameter field of a deferred word.

ASSIGN — cfa (executing) I
“assign” —(compiling)

Used to assign the action for a deferred word. Used in the form:

ASSIGN <action-word> TO-DO <deferred-word>

it returns, or compiles as a literal, the PFA of the word following in the input stream. See **TO-DO** and the chapters on multi-tasking and timers.

CRASH —
“crash”

The default contents of a word created by **DEFER**. If such a word is executed without another assignment to it being made, **CRASH** executes, and causes an error abort.

DEFER —
“defer”

Use in the form:

DEFER <name>

A defining word which creates a word called <name>. When <name> is executed, the contents of its parameter field are executed.

TO-DO cfa — (executing)
“to-do” — compiling

Uses the given cfa when executing, or compiles the code to use it when compiling. The result is that the given cfa is stored in the pfa of the word following in the input stream. Use in the form:

ASSIGN <action> TO-DO <name>

Blank page

Application Programs

7.0 Generating An Application Program

The word **SAVE** generates a file containing an dictionary image of the current Forth. Specifically, a file that can be passed as the first argument on the command line to Pinc PowerForth. **SAVE** sets the initial values of **DP FENCE** and **VOC-LINK** and preserves any other modified values in the start-up locations.

To generate a turnkey package that executes an application immediately, first compile the application, and then assign the required word to be used by **QUIT**. Assuming that the application word is called **APPLICATION** the following phrase will do the job:

ASSIGN APPLICATION TO-DO QUIT

If you want to modify the error handling change the assignment of **ABORT**. **ABORT** should reset the data stack, perform any error recovery, and then (according to the standard) should execute **QUIT**.

ASSIGN ERROR-HANDLER TO-DO ABORT

The standard action of **ABORT** is **(ABORT)** which is defined as:

```
: (ABORT)  ( — )
        S0 @  SP!
        QUIT
;
```

During the start-up of Pinc PowerForth almost the first word executed in **COLD** is the word **LOADER**. This is a deferred word used for the purpose of loading any required tools. Its default assignment is to execute **NOOP**. Before an application program can be generated, this assignment can be changed. If you do not wish to have the MPE Pinc PowerForth sign on message in your application, the action of **LOADER** can include **ABORT**, so bypassing the rest of **COLD**.

If the file to be saved is called <filename> the following will generate an executable file of that name. If **QUIT** and **ABORT** have been re-assigned, the new assignments will be used when the program starts.

SAVE <filename>

Where <filename> is a valid file name.

Blank Page

Pinc PowerForth Internals

8.0 Introduction

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

8.1 User Area Layout

The user area 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 are 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 <name> is later executed, the address returned by <name> is calculated by adding n to the base address of the user area, which is held in the processor’s DI register. The base address may be different for each task, or several tasks can share one user area.

⊗ Please note that at the time of going to press the locations given for user variables are correct, but may change in future releases of Pinc PowerForth.

The size of the User Area is set when Pinc PowerForth is created.

Summary Of The Use Of The User Variables

No. (Hex)	User Variable	Desciption
	WIDTH	Contains the maximum name field width. This word may well disappear in later versions of Pinc PowerForth.
	>IN	Contains the current offset from the start of the input stream.
	OUT	Contains the current output character position in the line.
	SPAN	Contains the number of characters last read into **TIB** by **EXPECT**.
	HLD	Is used during number conversion.
	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 length of the terminal input buffer.
	#L	Contains the number of cells last returned by **NUMBER?**.
	TAB-WIDTH	Contains the tab separation in characters.

#D @TABLEWORDS2 = 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.
	R0	Contains the offset of the top of the return stack in the task's stack segment.
	S0	Contains the offset of the top of the data stack in the task's stack segment.
	SYMBOL_TABLE	Reserved.
	WORD_BUFF	Buffer used by **WORD** to store text gleaned from the input stream. The address of this buffer is returned by **'WORD**.

PAD	Consists of two buffers: one is used for string handling, above **PAD**; and the other is used for numeric text conversion, immediately below **PAD**.
TIB	Terminal input buffer. This area is used to store lines read in by **EXPECT**.
INIT-FENCE	Initial value of the top of the dictionary. This variable is set using the **IS-FENCE** word to prevent words below the address placed within it being forgotten.
INIT-DP	Initial value of the dictionary pointer. This variable is set before the dictionary is saved to allow all words defned to date to be present when the dictioanary is reloaded.
INIT-VOC-LINK	Initial value of the vocabulary link. This variable is set to point to the last vocabulary defined. It is used to preserve the vocabulary structure defined when the dictioanry is being saved, for later use.
ROOT-VOC	Pointer to the first vocabulary defined in the system- **ROOT**.
CURRENT	Points to the vocabulary into which words are defined.
CONTEXT	Is an array of up to sixteen vocabularies which are searched in order. See the tutorial section on vocabularies.

8.2 Memory Map

8.3 Start Up Procedure

At program load the Forth virtual machine is started, and a startup word run. This word initialises some of the user variables, and runs **COLD**. **COLD** initialises the io, runs **LOADER** to load any required user setup action, and then runs **ABORT.**

ABORT resets the data stack and calls **QUIT**.

QUIT resets the return stack, and is the text interpreter main loop.

LOADER ABORT and **QUIT** are all deferred words, and may be set by the user to do any required action. Note that **LOADER** is initialised to do **NOOP**. Before an application program is generated and distributed, the assignment of **LOADER** can be

changed to start the application (although the return and data stacks will have to be initialised of course).

8.4 Pinc PowerForth Dictionary Structure

Pinc PowerForth uses a multiple thread vocabulary mechanism that gives extremely high compilation speeds. All Forth words use the same dictionary structure. For further information see the Power User Guide chapter.

Advanced Implementation and Extension.

This chapter details the advanced implementation techniques that can be applied to Pinc PowerForth and how the system can be extended. The chapter should be read by those with a good knowledge of both C and Forth who have already produced an implementation of Pinc PowerForth by following the “Backup and Installation” chapter.

9.0 Adding New Primitives.

This section describes how to add new primitives to Pinc PowerForth.

9.0.1 Forth Internal Definition Language (FIDL).

Unlike conventional Forth Systems, Pinc PowerForth primitives are not written in Assembly Language. Each Pinc PowerForth primitive which is said to be “understood” by the Pinc PowerForth virtual machine is written in a C-like language-Forth Internal Definition Language (FIDL). FIDL was chosen as opposed to straight C since it offers a terser syntax and concentrates the changes made to the virtual machine to one file called FIDL.SRC. This file is translated into a series of C source files which are included in various parts of the Pinc PowerForth program source code. This translation process is performed by the FIDL compiler. You will note the construction and execution of this compiler in the makefile.

9.0.2 FIDL Syntax.

The FIDL language has the following syntax.

<Forth Name> <Associated C Function name> <Type>
%

C Code for Primitive

%

The names and type are delimited by spaces

E.g. The following header defines the Forth word **;**, whose C function name is semicolon. The type of the word is described below:

; semicolon :I

A word may be any of three types:

V vocabulary

: colon definition
:I immediate colon definition.

The example below shows the FIDL code for the primitive **bye**. The reader will note that is a colon definitions whose Forth name is **bye** and whose C name is **goodbye**. The function prints out a message on the output stream and executes paren_bye.

```
bye goodbye :
%
   fprintf( output, “BYE” );
  paren_bye();
%
```

Comments may be included in this source file, these are lines which contain \ followed by a space in the first two columns.

The quote character “ should be escaped whenever it is used. Thus the header for the immediate forth word ”" would be:

```
\"\" quotes_quotes :I
```

9.0.3 Using the Stack Functions.

Most primitives you add to Pinc PowerForth will be required to manipulate the data and/or return stack. The file **stacks.c** provides a set of functions for pushing and popping items on/from both of these stacks. you may use these functions freely within the body of a primitive written in FIDL.

9.0.4 An Example: ROT.

An example of using these functions is the Forth word **ROT** which takes the third item down on the data stack and makes it the first. Using the stack functions the code to perform this operation is a follows:

```
rot rot :
%
   unsigned CELL data1, data2, data3;
   data1 = pop_data();
   data2 = pop_data();
  data3 = pop_data();
   push_data( data2 );
  push_data( data1 );
   push_data( data3 );
%
```

9.0.5 Addressing Ram.

In order to facilitate relocation of the Pinc PowerForth program in memory and reloading of saved dictionary files, there is no absolute addressing in Pinc PowerForth. All addresses refer to and array called **ram** and can be thought of as array indices. Thus to print the contents of byte 14 in ram as a character the statement would simply be:

```
printf( "%c", ram[ 14 ] );
```

Throughout the source code the reader will encounter the following phrase:

```
cell = (unsigned CELL *) &ram[ x ];
```

where x is some array index.

This code defines a **CELL** pointer "cell" into the array "ram". Thus to access and print a cell sized unsigned integer which starts at address 32 in ram we would write:

```
cell = (unsigned CELL *) &ram[ x ];
printf( "%ld", *cell );
```

9.0.6 An Example: Fetch @.

An example combining both the stack manipulation functions and the addressing of ram to read cells is the Forth word @ (fetch). The FIDL code to perform this operation is as follows:

```
pop_data();
   cell = (unsigned CELL *) &ram[ address ];
   push_data( *cell );
%
```

9.0.7 **Calling OS Libraries.**

You may call additional C functions from a standard library or and object module that you are going to bind in later. In the same way that you called the C stack manipulation functions. However, you should declare these functions as being extern in the file **prim.c**. This ensures that the C compiler passes the correct types of parameters to the function and returns the correct value (if any). If you are using a standard library, the extern declaration takes the form of including the appropriate header file of that library. This is shown in the example below.

9.0.8 An Example: Time.

An example of calling a standard library function is the Forth words **.TIME** which prints the time of day. The time of day may be found by calling the C functions gmtime and asctime. The FIDL source code for the function is therefore:

```
.time print_time :
%
   long time_now;
  struct tm *newtime;

   time( &time_now );
  newtime = gmtime( &time_now );
   printf( "%s\n", asctime( newtime ));
%
```

In addition we must edit the file **prim.c** to include the headers for the library functions time. The macro command to be added to **prim.c** is:

```
#include <time.h>
```

9.1 Adding New Secondaries.

This section describes how to add new secondaries to Pinc PowerForth.

9.1.1 CWeed Forth and BOOT.DIC.

When Pinc PowerForth is executed, it loads a image of the Ram array. This contains the User Variables and Dictionary Space of a complete Forth System. By default this file is called **boot.dic**, although this may be overridden by specifying a dictionary file argument on the command line. When a Pinc PowerForth system is made, a default version of **boot.dic** is created. This is done by a mini Forth System called CWeed Forth. You can follow the creation and execution of CWeed Forth in your makefile. CWeed Forth performs the following functions:

1) Builds the vocabularies and primitive dictionary headers of those specified in the **fidl.src** file.

2) Compiles a set of secondaries from CWeeds Secondary Code and Programming Environment (**cscape**) file.

It is customary for **cscape** to terminate with the phrases:

```
‘ cold 0 !
‘ warm 4 !

save boot.dic
bye
```

The above sets the Cold Boot Vector (address 0), Warm Boot Vector (address 4), saves of **boot.dic** and exits. You can change the contents of the vectors to execute you own application code if you wish, although we recommend that you use the “assign to-do” mechanism on the words **ABORT** or **LOADER** to ensure Pinc PowerForth is properly initialised.

9.1.2 Editing the CSCAPE File.

The **cscape** file is like any other Forth Source file. You can therefore add and remove Forth Source code to/from as with any other file. It contains all of the secondary words which make up full Pinc PowerForth forth. You can add customisations and system specifics to this file to taylor Pinc PowerForth to your requirements. E.g. you can specify the name of your favorite editor with the **EDITOR-IS** word so that Pinc PowerForth knows which editor to load when you type **ED**.

9.1.3 Creating a New BOOT.DIC.

When you have edited the **cscape** file you can build a new version of the **boot.dic** dictionary file. To do this simply perform a make.

9.2 Embedding Pinc PowerForth in an Application.

This section describes how to embed Pinc PowerForth in an existing C application to provide interactive debug and test capabilities on running C program. Before reading this section you should ensure you have experimented and feel confident with the concepts of Making Pinc PowerForth, CWeed Forth, Dictionary Saving and Boot Vectors.

9.2.1 Calling Bootstrap.

If Pinc PowerForth is embedded in an application it is unlikely that it will be the **main** program of the application. You are more likely to want to call Pinc PowerForth as a C function when a certain event occurs. E.g. the playing of keyboard chord or if an error occurs. At that point you will wish to “drop” into Pinc PowerForth to perform some debugging and write some test words.

To facilitate this type of behaviour Pinc PowerForth has been constructed as a C function which is called from **main**. The **main** program call a function **bootstrap** with an argument of the dictionary file it should load. It returns an exit code of success or error (some number other than 0). which is passed directly to the operating system via the **exit** function. The basic code for **main** is a follows:

```
main(argc, argv)
int argc;
char *argv[];
{
   if( argc<1 )
         exit( bootstrap( argv[1] ));
  else
          exit( bootstrap( DEFAULT_DICTIONARY_FILE ));
}
```

This is the only occurance of exit in the whole Pinc PowerForth system. All functions, such as **bye** return to **bootstrap** via a set jump, long jump scheme. Thus to call Pinc PowerForth as a function:

1) Edit a copy of **main.c** to remove the function main.

2) Edit a copy of the makefile so that is produces an object module of the complete system. Note: CWeed Forth must be compiled and run to produce a **boot.dic** file at least once.

3) Make a call to **bootstrap** in the appropriate place in you application and compile it.

4) Link your application with the object module of the Pinc PowerForth system.

In order to interact with your application you should add primitives to the **fidl.src** file which call relevant functions or address relevant data structures.

9.2.2 Booting Cold or Warm.

Using the above scheme, each time you call Pinc PowerForth from you application it will load a dictionary and Cold Start. This may be undesirable, since you may wish to call Pinc PowerForth several times from your application and not loose the state of Pinc PowerForth between calls. This can be achieved bye calling the **run_cforth** function instead of the **bootstrap** function on all subsequent entries to Pinc PowerForth. You will find this function in the source file **main.c**. REMEMBER **bootstrap** must be called first before subsequent calls to **run_cforth**.

The function **run_cforth** starts execution at the contents of the Cold Boot Vector (address 0). You may wish to change the contents of this vector to that of the Warm Boot Vector (address 4), before you leave Pinc PowerForth from the first call to **bootstrap**. This will allow the contents of the dictionary, I/O buffers and user variables to be preserved.

Inside Pinc PowerForth, The Power Users Guide.

This chapter is for Power Users of Pinc PowerForth. It describes the structure and behaviour of the complete Pinc PowerForth system in fine detail. In addition it provides a guide to the complete Pinc PowerForth C source code. You should be familiar with all previously defined concepts of Pinc PowerForth before you read this section. You should read this chapter with the accompanying source code manual.

10.0 Virtual Machine Structure.

This section describes the components that comprise Pinc PowerForth and how the implementation works. It assumes the reader is both familiar with C, Forth and the concepts of threaded interpreters.

10.0.1 Memory Areas.

The Pinc PowerForth virtual machine consists of four memory areas. These are: the data stack “S”, return stack “R”, general memory area “Ram” and user variable space “User Area”. All of these areas are modeled by C arrays. Their definition can be found in the file **main.c**. The size of these areas can be configured by modifying the values of the macros in the file **vmsize.h**.

The User Area overlaps the Ram area occupying in excess of the first 600 bytes. By providing this overlap, Forth users may apply the standard wordset to manipulate both Ram and the User Area. In addition, by making it impossible for the dictionary to begin at a very low address (below 600), machine primitives may be easily identified (see later).

10.0.2 Boot Vectors.

The first two cells before the User Area are reserved as cold and warm start vectors respectively. Thus when Pinc PowerForth boots the virtual machine begins to execute code at the address held in cell 0.

10.0.3 Additional Buffers.

The various buffer and table areas required by Forth are defined as part of the User Area. These include **TIB**, the address of the root vocabulary and current and context areas. The size of the context area is defined in the file **vocs.h**. Care should be taken when changing the size of this area to move the dictionary start accordingly.

The layout of the low ram area can be found in the file **lowram.h**.

10.0.4 Registers.

The virtual machine has the following registers:

sp Data Stack Pointer.

rp Return Stack Pointer.

dp Dictionary Pointer.

ip Instruction Pointer.

latest Pointer to the nfa of the last word defined

voc_link Pointer to the cfa of the previously defined vocabulary.

You will find these declared in the file **main.c** and imported into other files as required. Any of the registers may be declared as C **register** variables, rather than **auto d**epending upon your speed requirements and the processor on which Pinc PowerForth is running.

10.1 Forth Model.

The Pinc PowerForth model is a software implementation of a subroutine threaded Forth model, where the subroutine calls are implicit. This is similar to RTX processor range which implements the subroutine model in hardware. Thus a standard Forth word, compiled into Ram consists of a list of addresses representing the CFA of other words previously compiled in Ram. Such a word is termed a “Secondary”. An opcode or “Primary” does not present a CFA within Ram and is interpreted directly by the virtual machine. All colon definitions are terminated by the Primary NEXT.

10.1.1 Virtual Machine Primitives.

The Pinc PowerForth virtual machine “interprets” certain numeric values as opcodes. On encountering an opcode the virtual machine executes a C function which is associated with it. This function may manipulate the Ram, the stacks and/or perform some I/O.

10.1.2 Inner Interpreter and Next.

The execution of Forth programs is performed by one main C Function- The Inner Interpreter. The behaviour of this function is controlled by the C-Function Next. The pseudo code algorithms for the Inner Interpreter and Next are given below.

```
Proc Inner_Interpreter
   Forever:
      The cell pointed to by ip is checked to see if its
      contents are the opcode of a primitive.
      If they are
        ip is moved onto the next cell
        the primitive is executed.
      Else
        the address of the next cell is pushed onto the return stack
          ip is set to the contents of the cell


Proc Next
   ip = pop return stack;
```

Note this model is pre-incrementing: The address of the next cell in a word is placed on the return stack and not the current value of ip.

The complete source code of the Inner Interpreter can be found in the file **inner.c**. The Next opcode is defined in the file **fidl.src**.

10.1.3 What is a Pinc PowerForth Address?

In order to facilitate relocation of the Pinc PowerForth program in memory and reloading of saved dictionary files, there is no absolute addressing in Pinc PowerForth. All addresses refer to Ram and can be thought of as array indices.

Throughout the source code the reader will encounter the following phrase:

```
cell = (unsigned CELL *) &ram[ x ];
```

where x is some array index.

This code defines a CELL pointer “cell” into the array “ram”. Thus to access and print a CELL sized unsigned integer which starts at address 32 in Ram we would write:

```
cell = (unsigned CELL *) &ram[ 32 ];
printf( “%ld”, *cell );
```

10.1.4 Vocabulary Structure Mechanism.

The vocabulary mechanism used in Pinc PowerForth is a derivative of that proposed by Bill Ragsdale in the Forth-83 Standard. It is compatible with previous systems, while permitting control of the search order.

An 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 sixteen 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.

This singly linked, hashed dictionary structure, similar to other MPE products. Each vocabulary consists of 5 items. Theses are:

1) A standard header including a link field and name field.

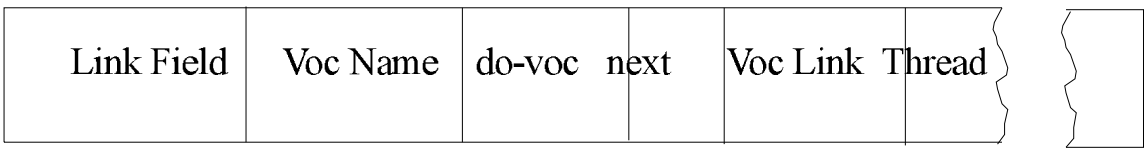
2) The primitive **do-voc** whose purpose is to put the cfa of the vocabulary into the context array.

3) The primitive next which terminates the execution of a vocabulary (after the **do-voc**).

4) A pointer to the cfa of the previous vocabulary.

5) The thread table of the vocabulary. This is an array of **MAX_THREADS** in length (see **vocs.h**).

The structure is shown in the diagram below:



When a word is compiled into a vocabulary the name field of the word is hashed into a number between 0 and **MAX_THREADS**-1 by taking the first char of the name and its length and **anding** this with **MAX_THREADS**-1. The hash value forms an index into the vocabularies thread table. Each entry in the thread table is a linked list of words in that vocabulary which have the same hash value. The new words **LINK FIELD** is added to the front of the list. N.B. the list front of the list is always the item in the thread table.

10.1.5 Word Structure.

A standard colon definition in Pinc PowerForth consists the following, all of which are cell aligned:

1) A link field which points to the link field of the previous word in the same thread (see above).

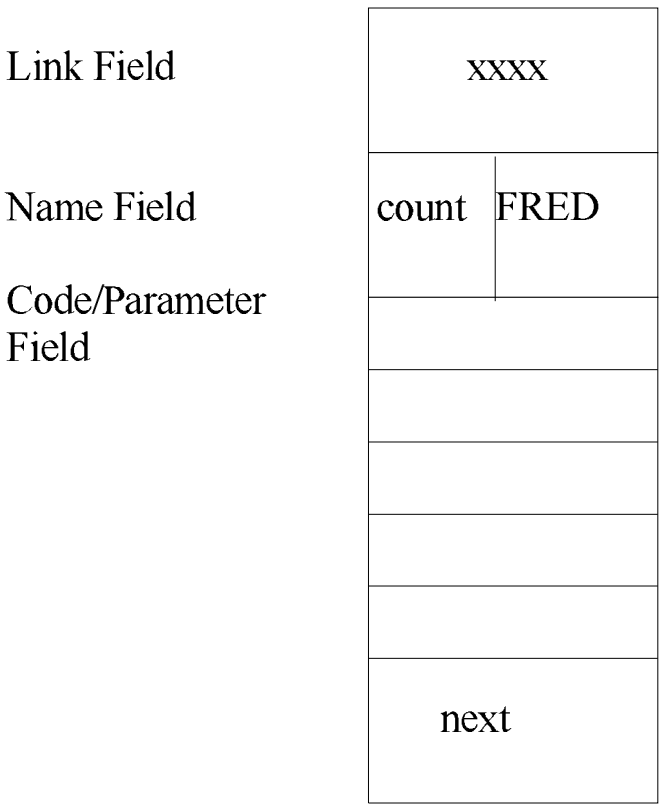
2) A name field which contains both a count byte and a terminating trailing zero for C's benefit. The bottom five bits of the count byte contain the actual count, and the top three bits are used as follows:

Bit 7: Always set
Bit 6: Set if word is **IMMEDIATE**
Bit 5: Set if word is hidden from search (**SMUDGE**d)

3) A word aligned CFA. This does not contain a jump to an inner interpreter or a **do-colon** primitive. It contains either the CFA of another secondary, or the opcode of a primitive instruction. All **do-colon** actions are implicit.

4) A PFA consists of one or more CFAs or primitive opcodes and is always terminated by the primitive opcode **next**.

The structure of word is shown in the diagram below:



Blank Page

The Pinc PowerForth C Source Code Guide.

This chapter provides the reader with a guide to the structure of the Pinc PowerForth source files. Each file is listed with a brief description of its contents and function.

11.0 The .H files.

11.0.1 FNAME.H

This file defines the filenames of some of the standard file used by Pinc PowerForth. E.g. the name of the Pinc PowerForth error file.

11.0.2 LOWRAM.H

This file describes the Pinc PowerForth Startup Vectors, User area, Current and Context Tables and where the dictionary starts above it. The first two cells of ram are defined to be the cold and warm startup addresses of Pinc PowerForth. These should be set to the CFAs of the words COLD and WARM. Alternatively the user can set them to point to their own application. If their contents are zero then Pinc PowerForth knows it has not had the secondary level of code compiled on top of it. This is useful for the error handler, which will either jump through warm if full Pinc PowerForth is running or just return after printing the error in CWeed Forth.

TIB is defined to be very large in order for buffered file I/O using read and write to be implemented at some later stage.

NOTE: It is important that the dictionary start address is greater than the ordinal value of the last Pinc PowerForth Opcode. This property is used by Pinc PowerForths inner interpreter to determine whether it is dealing with either a secondary or a primitive.

11.0.3 MISC.H

This file contains misc Constants and Macros.

11.0.4 NFMASKS.H

This file defines the bit patterns associated with the count byte of the Name Field of a Pinc PowerForth definition. The count byte is distinguished as being at the start of the name string by the fact that its top bit is set (bit7). If the next bit is set (bit6) then the word is defined to be immediate. If the next bit is set (bit5) then the word has been smudged. To Enable Fast thread table searches the Inverse of the Immediate and nfa masks is included.

11.0.5 OS.H

This file defines the Operating System Specific Parts of Pinc PowerForth Note the space at the end of the some of the command strings is significant.

11.0.6 STRSIZES.H

This file defines the size of the strings used by Pinc PowerForth.

11.0.7 STRUCT.H

This file associates a unique number with the type of control structure that may be being compiled by Pinc PowerForth.

11.0.8 VMSIZE.H

Describes the sizes of the stacks and memory of the Pinc PowerForth Virtual Machine.

11.0.9 CMODEL.H

This file describes the type of C Model you are compiling Pinc PowerForth on-Large/Small. In addition it also defines the C Compiler you are using for any implementation specific parts of Pinc PowerForth.

11.0.10 VOCS.H

This file defines the number of vocabularies that context can contain and the number of threads that are contained in any vocabularies thread table.

11.0.11 CFORTH.H

Header included by all Pinc PowerForth Source Files

11.1 The .C files.

11.1.1 THREADS.C

This file contains the set of functions for traversing Pinc PowerForths Vocabulary Thread Tables.

11.1.2 CASECON.C

Pinc PowerForth can be implemented to form headers in either upper or lower case. It can even be made case sensitive if you want. These two functions can be used to convert strings into either upper or lower case.

11.1.3 INNER.C

This file contains the Pinc/CWeed Forth inner interpreter.

11.1.4 DICLOAD.C

This file contains the functions for loading a dictionary file of a given name from disk.

11.1.5 DICMANIP.C

The following words manipulated the dictionary pointer and store data into ram.

11.1.6 IO.C

This file contains the Pinc PowerForth IO manager.

11.1.7 FIDL.C

The Forth primitives are defined in FIDL (Forth Internal Definitions Language). This program takes input form the fidl.src file and produces the following fields which are included in the Pinc PowerForth C source.

enum.h	An enumerated type for primitives.
proto.h	The C Function prototypes.
extproto.h	The C Function prototypes as external declarations.
prim.h	The C Function forth machine primitives.
carve.h	The C Function calls to carve dictionary headers.
assign.h	The assignments of function pointer to the execution vector.

11.1.8 HEADTRAV.C

The following functions convert given address fields to other address fields.

11.1.9 MAIN.C

Definition of the Forth Virtual Machine

11.1.10 PRIM.C

The functions which define the Pinc PowerForth primitives are inserted here. These are generated by the FIDL compiler and can be found in the file prim.h. In addition the function assign_primitives assigns their addresses into the execution vector.

11.1.11 N2STR.C

This file contains a set of word for converting number into strings. In addition the function print_one_stack line is provided for use by .s

11.1.12 CWEED.C

This file contains the CWeed Forth Interpreter, Compiler and Initialisation Functions.

11.1.13 ERROR.C

This file contains the Pinc PowerForth Error Handler.

11.1.14 STR2N.C

This file contains the functions used by Pinc PowerForth/Cweed Forth to convert a string into a number depending on the base.

11.1.15 STACKS.C

The following functions define the basic operation that can be performed upon the data and return stacks. At present these functions are used extensively by the Pinc PowerForth primitives and no manipulation of the array structures representing the stacks is allowed except through these functions. Thus the code for the Pinc PowerForth primitives is specification of the stack manipulation involved but is not efficient in terms of execution. Later version of Pinc PowerForth may define primitives to explicitly manipulate the arrays and registers which define the stacks. By use of a #inline compiler directive the stack manipulation functions may be inserted inline inside Pinc PowerForth primitives. This will allow optimising compilers to remove redundant variable assignments and manipulations.

11.1.16 CARVE.C

This file contains the set of functions which carve headers, compile colon definitions and make vocabularies.

References And Bibliography

12.0 **References**

To keep in touch with the Forth community, and to take advantage of other people's wisdom and experience, it is invaluable to subscribe to one or more of the user magazines.

In Europe try:

MicroProcessor Engineering,
133 Hill Lane
Shirley
Southampton SO1 5AF
England

tel: UK: . . . . . (0703) 631441
International: . . . (+44) 703 631441

In America try:

Forth Interest Group
PO Box 8231
San Jose
California CA 95155

tel: (+01) 408-277-0668

or:

Mountain View Press
PO Box 4656
Mountain View
California CA 94040

tel: (+01) 415-961-4103

12.1 **Bibliography**

An ever increasing number of books about Forth is available for all dialects of Forth. Most Forth programmers end up buying several before finding enough to cover all aspects properly.

All of the following books can be obtained from ‘Good Bookshops’ and MPE directly, see our catalogue for latest prices.

12.1.1 Forth Text & Reference Books

:

& ( )

A very large (more than 450 pages) text book on Forth. It includes chapters on most subjects, the chapter on defining words does not fudge the issue (very rare), and it has a large collection of examples with answers (as source code) in the back. Based on Forth-83. Probably the most complete of the textbooks.

( )

The classic book on Forth, it is not only well written, it is accurate, useful, contains many examples of everyday Forth usage, and is complemented by a selection of problems with answers. Now into the second (improved) edition. Every Forth programmer ends up buying at least one of Brodie’s books.

( )

The second of the Leo Brodie books, this one is about the philosophy and practice of Forth. It includes quotes from senior Forth programmers, discussions of Forth project management, as well as an introduction to its more advanced features. Recommended

( )

Really a book about defining words. The overall theme being using defining words to create an object oriented extension to Forth. The book is based on the implementation of abstract data types in Forth, and the creation of systems using object based techniques, the book explores defining words and vocabulary techniques. Well written, and a valuable source of ideas. Recommended as an excellent second level book about Forth.

&

This provides standard tools for programmers with a working knowledge of Forth who need to develop tools. The book includes sections on debugging techniques, sorting and merging, binary searches, and floating point. To Forth-79 standard, with references to fig-Forth. Recommended.

An introduction to Forth for people who have work to do. Based around small applications, this book will probably suit many practical programmers. Liked by a customer who has evaluated nearly all the Forth books available.

:

If you are starting out in the field of real time control of real hardware under Forth, then this book is for you! This text covers most aspects of real time control under Forth, from the very basics of what Forth is, through to control loops and digital implentations of analogue filters.

A specification for the layout of Forth source code. Generated by the BT Martello team at Martlesham as a result of training Forth programmers, it aims to provide an outline that can be followed by newcomers to Forth, as well as providing standards for experienced Forth users.

( )

A good introductory book, readable and well regarded. Contains everything the new user will need. The emphasis is on Forth-79, but valuable regardless of the dialect being used. Linda’s favourite.

. , .

This text descibes the evolution of Stack Computers and their impact on the computing and industrial controller markets. Written by one of the chief designers behind the RTX range of products, this book describes all major aspects of stack machine architecture and its impact on software implementation and efficiency.

The subject of this book is threaded code systems (TILs), and how to build one from scratch, using Forth on a Z80 as an example. Now somewhat dated, but if you are interested in other forms of Forth interpreters, or have to write a new Forth from the ground up, this book contains much of value.

12.1.2 Non-Forth Books

( )

The best of the technical guides to programming on the PC. Now in its second edition.

12.1.3 Papers & Proceedings

. ( )

( )

1987

1988

FORML is the main US West Coast theoreticians conference, and Rochester is more industrial conference on the East Coast. Their proceedings contain much of interest to professionals, tool-builders, and language implementors. JAFAR is a refereed technical journal, the bibliography contains all known references to Forth at the time of publication. The Index is a guide to the contents of FORML, Rochester, and JAFAR.

83

12.1.4 Newsletters

( )

The US Forth user group. The original Forth magazine and still the best. The quality of some of the articles is high. FIG take credit cards, and back issues are available.

( )

. .

The magazine of the English Forth Interest Group chapter. Available from:

Membership Secretary - fig (UK)
54 Wild Briar
Wokingham
Berkshire RG11 4UL

Blank page

Error Messages

13.0 Introduction

Pinc PowerForth error messages are stored in text form in the screen file **pinc.err**, where the message number n corresponds to the n-th line in the file (n starting with 0). If the file cannot be opened, an error number is given. Many of the error numbers are a hangover from the original fig-Forth standard. Not all of the error messages originally defined are appropriate to Pinc PowerForth, and some error messages and numbers were not assigned by the standard. Consequently not all numbers are used. The file **pinc.err** is a normal text file, and can be edited by you standard editor if new messages need to be added, or the text of an existing message needs to be changed. This file should be kept in the same directory as the one in whic you are running.

13.1 Forth Compiler/Interpreter Errors & Messages

0

The word does not exist in the dictionary. Often caused by mistyping the name of the word.

1 ,

The parameter stack is empty. Caused by taking too many items off the stack.

2 ,

The dictionary space is exhausted. Usually caused by either running out of space by loading too much code.

3

4

The word being defined already exists. This is a warning only.

5

The word does not exist in the dictionary. Often caused by mistyping the name of the word.

7

The parameter stack space is exhausted. Usually caused by a software fault. Try looking for a loop that leaves too many items on the stack.

8 ? ?

9

12

A deferred word has not had an assignment made to it. When a deferred word is created, its initial assgnment is the word **CRASH** which causes this error message.

13

The current value of base is not decimal.

14

17 , ,

A word which can be used only inside a colon definition has been used outside a colon definition. Many of Forth's structure words are immediate, and can only be used when compiling.

18

The word has been used during compilation. Usually this means that "**:**" has been used inside a definition, or the last definitions' "**;**" was missing.

19

Conditional words are not paired or nested correctly. Caused either by a stack fault from an immediate word inside a conditional structure, or caused by incorrectly overlapped conditional structures, e.g:

IF ... BEGIN ... ELSE ... UNTIL ... ENDIF

20

A definition has been terminated by a semicolon before it has been completed. Usually caused by not finishing a conditional structure properly.

21

An attempt has been made to **FORGET** below **FENCE**, which marks the upper limit of the protected dictionary.

22 ,

The operation —> was executed from the keyboard, not from a screen being loaded from disc.

23 0..32767 (0..7)

An attempt has been made to edit a screen outside the screen's bounds. Only relevant when using the old fig-FORTH line editor.

24

An attempt to **FORGET** has been made when the **CONTEXT** and **CURRENT** vocabularies are not the same.

25

13.2 Assembler Errors

33

An attempt has been made to use an addressing mode, or combination of modes, that is invalid for the instruction.

34 8 080..07

Byte offsets in 80x86 instructions are signed, and so are restricted to the range -080h..07Fh. The word value to be truncated was checked and found to be outside this range.

35

The required number of shifts is too large for this instruction, or this form of the instruction.

45

The local label facility has a limited number of labels, and the required label was out of this range.

46

A reference to a local label was made in this word, and has not been resolved (the label does not yet exist) at the end of the word.

47 ,

This is the result of a catch-all check performed by **END-CODE**. The stack depth on exit is not the same as on entry to the code definition. This usually means that something untoward has happened during address arithmetic.

13.3 Memory Handling Errors

52

Pinc PowerForth asked DOS for more memory, and couldn't get it. This error will occur if there simply isn't enough memory in the computer, or if the memory allocation is too fragmented to provide a hole big enough.

54 49

A request to DOS was made to release a block of memory that DOS doesn't know about. This is either a simple programming error or a total disaster if an amok program has trashed the linked list of memory block headers.

13.4 Block/Screen File I/O Errors

65
No screen file was open when the operation needed one.

66 /
The file pointer could not be set to point at the required block number.

67 /
A fatal error occurred while writing to the file.

68 /

69

70

72

73

74

75

76

77 /

78

13.5 MSDOS Errors

81

DOS doesn't know what to do as the function number is undefined.

82

83 ? ?

84

All handles are in use. DOS normally enforces a limit of 20 open handles. It is possible to get round this. Contact MPE for details.

85 . .

86 / ?

87 !

The DOS memory header list has been trashed.

88

640k/1Mb is not enough

89

90

Previous SET or PATH command bad, or the environment has been trashed.

91

92

The required access code does not mean anything to DOS. Either a simple programming error, or the application needs a higher DOS version than the one being used.

93

95

Attempt to access a drive that does not exist in this system.

96

97

98

13.6 Text File Inclusion Errors

113

114

115

116

117

118

119

120

13.7 Errors Specific to Pinc PowerForth

145 .

146 + ? .

147 .

148 .

149 .

150 .

151 ..

152 .

153 .

154 .

155 .

156 .

157 .

158 .

159 .

160 .

Glossary Notation

The documentation of the glossaries uses a methodology based on that used for the FORTH-83 Standard document. As this is not a standard document, but a user manual, we have taken some liberties to make this manual easier to look at.

We cannot call the Glossary a Dictionary, as this is a Forth term for the system itself.

You may come across some words in the dictionary which are not documented. These words are undocumented because they are words which are only used in passing as part of other words; also these words may not exist in later versions of PC PowerForth, and their existence should not be relied upon.

14.0 Order

The glossary definitions are listed in the following order:-

, . : ; ! ? “ < ‘ ( ) [ ] { } $ + - * / ^ = % # & @ \ _ | > 0..9 A..Z

14.1 Forth Words

Word names in text are capitalized and bolded throughout, e.g. the Forth word **SWAP**.

Forth program examples are shown in a courier font thus:

```
: NEW-WORD  ( — )
        OVER DROP
;
```

14.2 Stack Notation

The stack parameters input to and output from a definition are described using the notation:-

before — after

where: ‘before’ means the stack parameters before execution
and ‘after’ means stack parameters after execution.

In this notation, the top of the stack is to the right. Words may also be shown in context when appropriate. Unless otherwise noted, all stack notation describes execution time. If it applies at compile time, the line is followed by:

(compiling)

14.3 Stack Parameters

Unless otherwise stated all references to numbers apply to 16-bit signed integers. The implied range of values is shown as {from..to}. The content of an address is shown by double braces, particulary for the contents of variables, e.g., BASE {2..72}. The following are the stack parameter abbreviations and types of numbers used throughout the glossary. These abbreviations may be suffixed with a digit to differentiate multiple parameters of the same type.

STACK ABBRV.	NUMBER TYPE	RANGE IN DECIMAL	MINIMUM FIELD
flag	boolean	0=false, else=true	16
true	boolean	1 (as a result)	16
false	boolean	0	16
b	bit	{0..1}	1
char	character	{0..127}	7
8b	8 bits	not applicable	8
16b	16 bits	not applicable	16
n	number	{-32,768..32,767}	16
+n	+ve int	{0..32,767}	16
u	unsigned	{0..65,535}	16
w	unspecified weighted number (n or u)	{32,768..65,535}	16
addr	address	{0..65,535}	16
32b	32 bits	not applicable	32
d	double number	{-2,147,483,648 ..2,147,483,647}	32
+d	positive double number	{0..2,147,483,647}	32
ud	unsigned double number	{0..4,294,967,295}	32
d	unspecified weighted double number (d or ud)	{-2,147,483,648.. 4,294,967,295}	32
sys	0, 1, or more system dependent entries	n/a	n/a

Any other symbol refers to an arbitrary signed 16-bit integer in the range {-32,768..32,767}, unless otherwise noted. Because of the use of two's complement arithmetic, the signed 16-bit number (n) -1 has the same bit representation as the unsigned number (u) 65,535. Both of these numbers are within the set of unspecified weighted numbers (w). On many occasions where the context is obvious, informal names are used to make the documentation easier to use.

14.4 Input Text

<name>

An arbitrary Forth word accepted from the input stream. This notation refers to text from the input stream, not to values on the data stack.

ccc

A sequence of arbitrary characters accepted from the input stream until the first occurrence of the specified delimiter character. The delimiter is accepted from the input stream, but it is not one of the characters ccc and is therefore not otherwise processed. This notation refers to text from the input stream, not to values on the data stack. Unless noted otherwise, the number of characters accepted may be from 0 to 127.

14.5 Other markers

The following markers may appear after a word's stack comment. These markers indicate certain features and peculiarities of the word.

C The word may only be used during compilation of a colon definition.

I The word is immediate. It will be executed even during compilation, unless special action is taken, e.g. by preceding the word with the word **[COMPILE]**.

U A user variable.

Blank page

Licence Terms

16.0 Distribution Of Application Programs

Providing that the user can have no access to the underlying Forth and its text interpreter, applications written in Pinc PowerForth may be distributed without royalty. An acknowledgement will be gratefully appreciated.

Distributed applications may be based on the file **pinc** and any number of overlays. Object code generated from the source files can of course be included in your applications. MPE source files and all other files are part of the development environment, which may not be distributed without prior permission in writing from MicroProcessor Engineering.

16.1 **Pinc PowerForth source code**

FORTH development packages are available for software developers. These packages allow code to be placed in ROM for dedicated applications, and/or to be compiled without dictionary headers for space or security reasons. The packages include a cross compiler with full source code, and source code for a FORTH nucleus.

MPE Forth cross compilers support more than a dozen different processors, including the 80x86, and 680x0 families. The Z80, 80x86 and 680x0 targets include a nucleus compatible with Pinc PowerForth. Software developers can apply for all or part of the source code of the current version of Pinc PowerForth.

16.2 Warranties, Support, and Copyright

We try to make Pinc PowerForth as good as we possibly can. We support our products. If you find a bug in Pinc PowerForth 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 however, contact us or your supplier first in case the problem has already been fixed.

Make as many copies as you need for backup and security. The discs are not copy protected. It is copyrighted material and only **ONE** copy of it should be in use at any one time. Contact ourselves or your vendor for details of multiple copy terms and site licensing.

As we sell copies of Pinc PowerForth through dealers and purchasing departments we cannot keep track of all our users. If you fill out the registration card enclosed, and send it back to us, we will put you on our mailing list. This way we will be able to keep you informed of updates and new extensions for Pinc PowerForth. If you want direct technical support from us we will need these details to respond to you. If you have lost the registration card that came with the manual, please return the following form to us. 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.

16.3 Software Registration Form

SEND TO:

MicroProcessor Engineering Limited
133 Hill Lane
Shirley
Southampton SO1 5AF
England

Tel: (+44) 703 631 441
Fax: (+44) 703 339 691

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. Overseas customers - please include the country with your address!

Name:

Company:

Department:
@FILL IN ... = Address:

Telephone number & extension:

Package details

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

Title:

Computer type: (PC,XT,AT,386,Sun,Vax; clone; etc)

Operating system: (MSDOS,PCDOS,Unix,VMS; version)

Disc format: (5 1/4",3 1/2"; 360k,720k; etc)

Serial number:

Purchase date:

Supplier:

Blank page

Index

!

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X