Common Target Code v6.2

MicroProcessor Engineering

September 17, 2004

Contents

1 High level kernel KERNEL62.FTH **1**

1.1 User variables 1
1.2 System Constants 3
1.3 System VARIABLEs and Buffers 3
1.3.1 Variables 3
1.4 Deferred words 4
1.5 Predefined Vocabularies 4
1.6 Vectored I/O handling 4
1.6.1 Introduction 4
1.6.2 Building a vector table 4
1.6.3 Generic I/O words 5
1.7 String and memory operations 6
1.8 Dictionary management 6
1.9 String compilation 8
1.10 Pre-ANS Exception handlers 9
1.11 ANS words CATCH and THROW 9
1.11.1 Example implementation 10
1.11.2 Example use 11
1.11.3 Gotchas 11
1.12 Formatted and unformatted i/o 12
1.12.1 Setting number bases 12
1.12.2 Numeric output 12
1.12.3 Numeric input 13
1.13 String input and output 14
1.14 Source input control 15
1.15 Text scanning 16
1.16 Miscellaneous 17

1.17 Wordlist control 18
1.18 Control structures 19
1.19 Target interpreter and compiler 21
1.20 Compilation and Caches 24
1.21 Startup code 24
1.21.1 Cold chain 24
1.21.2 The COLD sequence 25
1.22 Kernel error codes 25
1.23 Differences between the v6.1 and 6.2 kernels 26
1.23.1 Error handling 26
1.23.2 Terminal input buffer and ACCEPT. 27

2 Character Queues 29

3 Heap Memory Allocation 31
3.1 Heap definition 31
3.1.1 16 bit targets - HEAP16.FTH 31
3.1.2 32 bit targets - HEAP32.FTH 32
3.2 Gotchas 32
3.3 Glossary 32
3.4 Diagnostics 33

4 Target VALUE and local variables 35

5 Target local variables 37

6 Ethernet and IP devices generic I/O 39
6.1 Internet Protocol Devices 39

7 Development tools 41

8 Debugging tools 43
8.1 Implementation dependencies 43
8.2 Miscellaneous 45
8.3 Stack checking 47
8.4 Assertions 48

9 ANS Environment System 51

10 Software Floating Point **53**

10.1 Introduction . . . 53
10.2 Source code . . . 53
10.3 Entering floating-point numbers . . . 54
10.4 The form of floating-point numbers . . . 54
10.5 Creating variables . . . 54
10.6 Accessing variables . . . 54
10.7 Creating constants . . . 54
10.8 Using the supplied words . . . 55
10.8.1 Calculating sines, cosines and tangents . . . 55
10.8.2 Calculating arc sines, cosines and tangents . . . 55
10.8.3 Calculating logarithms . . . 55
10.8.4 Calculating powers . . . 55
10.9 Degrees or radians . . . 56
10.10 Displaying floating-point numbers . . . 56
10.11 Changes from v6.0 to v6.1 . . . 56
10.11.1 32 bit targets: software floating point . . . 56
10.11.2 16 bit targets: software floating point . . . 57
10.12 Glossary . . . 57
10.12.1 Basic stack and memory operators . . . 57
10.12.2 Floating point defining words . . . 58
10.12.3 Type conversions . . . 58
10.12.4 Arithmetic . . . 59
10.12.5 Relational operators . . . 60
10.12.6 Rounding . . . 60
10.12.7 Miscellaneous . . . 61
10.12.8 Floating point output . . . 61
10.12.9 Floating point input . . . 63
10.12.10 Trigonmetric functions . . . 64
10.12.11 Power and logarithmic functions . . . 65
10.13 High Level primitives . . . 66

11 Periodic Timers **67**

11.1 The basics of timers . . . 67
11.2 Considerations when using timers . . . 68
11.3 Implementation issues . . . 68
11.4 Timebase glossary . . . 69

12 Vocabulary and wordlist tools **71**

13 XMODEM Receiver and Transmitter **73**

13.1 Introduction 73
13.2 Words in XMODEMTXRX.FTH 73
13.2.1 Configuration 73
13.2.2 Constants and variables 73
13.2.3 Common code 74
13.2.4 XMODEM transmission 74
13.2.5 XMODEM reception 75

14 ROM PowerForth utilities **77**

14.1 Introduction 77
14.2 Compiling text files 77
14.2.1 The required files 78
14.2.2 Compiling a specified text file 78
14.3 Downloading a binary image 78
14.3.1 XMODEM binary image download 78
14.3.2 Intel hex download 79
14.4 ROM PowerForth 79
14.4.1 Hardware requirements 79
14.4.2 EPROM/Flash area 79
14.4.3 RAM area 79
14.4.4 RAM/EPROM area 80
14.4.5 Types of board 80
14.4.6 Making your application turnkey 80
14.4.7 Discarding the application RAM area 81
14.4.8 Changing the application RAM start address 81
14.4.9 AIDE file server protocols 81
14.5 IODEF.FTH 81
14.5.1 AIDE support 81
14.6 Miscellaneous 82
14.7 Application Extensions 82
14.8 INCLUDE source code from AIDE 83
14.9 Simple source file loader 84
14.10Intel Hex transfers 84
14.11Block support 84

14.11.1 Primitives 85
14.11.2 Application words 85
14.11.3 Block file management 86
14.12 Target BUFFER: and VARIABLE 87
14.13 Some simple tools 87

15 Examples directory **89**
15.1 Main directory 89
15.2 Contributions subdirectory 90
15.3 Drivers subdirectory. 90
15.4 I2C subdirectory 91
15.5 SPI subdirectory 91

Chapter 1

High level kernel KERNEL62.FTH

1.1 User variables

`variable next-user \ -- addr`

Next valid offset for a USER variable created by +USER.

`: +user \ size --`

Creates a USER variable size bytes long at the offset given by NEXT-USER and updates NEXT-USER.

`tcb-size +user SELF \ task identifier and TCB`

When multitasking is enabled by setting the equate TASKING? the task control block for a task occupies TCB-SIZE bytes at the start of the user area. Thus the user area pointer also acts as a pointer to the task control block.

`cell +user S0 \ base of data stack`

Holds the initial setting of the data stack pointer. N.B. S0, R0, #TIB and 'TIB must be defined in that order.

`cell +user R0 \ base of return stack`

Holds the initial setting of the return stack pointer.

`cell +user #TIB \ number of chars currently in TIB`

Holds the number of characters currently in TIB.

`cell +user 'TIB \ address of TIB`

Holds the address of TIB, the terminal input buffer.

`cell +user >IN \ offset into TIB`

Holds the current character position being processed in the input stream.

`cell +user XON/XOFF \ true if XON/XOFF protocol in use`

True when console is using XON/XOFF protocol.

`cell +user ECHOING \ true if echoing`

True when console is echoing input characters.

`cell +user OUT \ number of chars displayed on current line`

Holds the number of chars displayed on current output line. Reset by CR.

`cell +user BASE \ current numeric conversion base`

Holds the current numeric conversion base

`cell +user HLD \ used during number formatting`

Holds data used during number formatting

`cell +user #L \ number of cells converted by NUMBER?`

Holds the number of cells converted by NUMBER?

`cell +user #D \ number of digits converted by NUMBER?`

Holds the number of digits converted by NUMBER?

`cell +user DPL \ position of double number character id`

Holds the number of characters after the double number indicator character. DPL is initialised to -1, which indicates a single number, and is incremented for each character after the separator.

`cell +user HANDLER \ used in catch and throw`

Holds the address of the previous exception frame.

`cell +user OPVEC \ output vector`

Holds the address of the I/O vector for the current output device.

`cell +user IPVEC \ input vector`

Holds the address of the I/O vector for the current input device.

`cell +user 'AbortText \ Address of text from ABORT"`

Set by the run-time action of ABORT" to hold the address of the counted string used by ABORT" <text>"

`#64 chars dup +user PAD`

Holds the

1.2 System Constants

Various constants for the internal system.

FALSE The well formed flag version for a logical negative

TRUE The well formed flag version for a logical positive

BL An internal constant for blank space

C/L Max chars/line for internal displays under C/LINE

#VOCS Maximum number of Vocabularies in search order

VSIZE Size of CONTEXT area for search order

XON XON character for serial line flow control

XOFF XOFF character for serial line flow control

1.3 System VARIABLEs and Buffers

1.3.1 Variables

Note that FENCE DP and VOC-LINK must be declared in that order.

WIDTH maximum target name size

FENCE protected dictionary

DP dictionary pointer

VOC-LINK links vocabularies

RP Harvard targets only. The equivalent of DP for DATA space.

SCR If BLOCKS? true; for mass storage

BLK If BLOCKS? true; user input dev: 0 for keyboard, >0 for block

CURRENT Vocabulary/wordlist in which to put new definitions

STATE Interpreting=0 or compiling=-1

CSP Preserved stack pointer for compile time error checking

CONTEXT Search order array

LAST Points to name field of last definition

#THREADS Default number of threads in new wordlists

1.4 Deferred words

`defer NUMBER? \ addr -- d/n/- 2/1/0`

Attempt to convert the counted string at 'addr' to an integer. The return result is either 0 for failed, 1 for a single-cell return result (followed by that cell) or 2 for a double-cell return. The ASCII number string supplied can also contain implicit radix over-rides. A leading $ enforces hexadecimal, a leading # enforces decimal and a leading % enforces binary. Hexadecimal numbers can can also be specified by a leading '0x' or trailing 'h'. When one of the floating point packs is compiled, the action of NUMBER? is changed.

`defer ERROR \ n -- ; error handler`

The standard error handler reports error n. If the system is loading, the offending line will be displayed. Now implemented by default as a synonym for THROW. Removed from v6.2 onwards.

1.5 Predefined Vocabularies

FORTH Is the standard general purpose vocabulary

ROOT This vocabulary stores the bare minimum functions

1.6 Vectored I/O handling

1.6.1 Introduction

The standard console Forth I/O words (KEY?, KEY, EMIT, TYPE and CR) can be used with any I/O device by placing the address of a table of xts in the USER variables IPVEC and OPVEC. IPVEC (input vector) controls the actions of KEY? and KEY, and OPVEC (output vector) controls the actions of EMIT, TYPE and CR. Adding a new device is matter of writing the five primitives, building the table, and storing the address of the table in the pointers IPVEC and OPVEC to make the new device active. Any initialisation must be performed before the device is made active.

Note that for the output words (EMIT, TYPE and CR) the USER variable OUT is handled in the kernel before the funtion in the table is called.

1.6.2 Building a vector table

The example below is taken from an ARM implementation.

```
create Console1      \ -- addr
  ' serkey1i ,             \ -- char
  ' serkey?1i ,            \ -- flag
```

```
  ' seremit1 ,              \ char --
  ' sertype1 ,              \ c-addr len --
  ' serCR1 ,                \ --

Console1 opvec !  Console1 ipvec !
```

1.6.3 Generic I/O words

: `KEY? \ -- flag ; check receive char`

Return true if a character is available at the current input device.

: `KEY \ -- char ; receive char`

Wait until the current input device receives a character and return it.

: `EMIT \ -- char ; display char`

Display char on the current I/O device. OUT is incremented before executing the vector function.

: `TYPE \ caddr len -- ; display string`

Display/write the string on the current output device. Len is added to OUT before executing the vector function.

: `CR \ -- ; display new line`

Perform the equivalent of a CR/LF pair on the current output device. OUT is zeroed before executing the vector function.

: `TYPEC \ caddr len -- ; display string`

Display/write the string from CODE space on the current output device. Len is added to OUT before executing the vector function. N.B. Harvard targets only. In non-Harvard targets, this is a synonym for TYPE.

: `SPACE \ --`

Output a blank space (ASCII 32) character.

: `SPACES \ n --`

Output 'n' spaces, where 'n' > 0. If 'n' < 0, no action is taken.

: `FlushKeys \ --`

Compiled for 32 bit systems to flush any pending input that might be returned by KEY.

: `SetConsole \ device --`

Sets KEY and EMIT and frieds to use the given device for terminal I/O. Compiled for 32 bit systems, but is also part of LIBRARY.FTH.

1.7 String and memory operations

Some of these words may be coded for performance. If they are predefined, the high level versions will not be compiled.

For byte-addressed CPUs (nearly all except DSPs) this kernel assumes that a character is an 8 bit byte, i.e. that:

```
char = byte = address-unit
```

```
: PLACE \ c-addr1 u c-addr2 -- ; copies uncounted string to counted
```

Place the string c-addr1/u as a counted string at c-addr2.

```
: BOUNDS \ addr len -- addr+len addr
```

Modify the address and length parameters to provide an end-address and start-address pair suitable for a DO ... LOOP construct.

```
: upc \ char -- char' ; convert to upper case
```

If char is in the range 'a' to 'z' convert it to upper case. Note that this word is language specific and is written to handle English only.

```
: UPPER \ c-addr u --
```

Convert the ASCII string described to upper-case. This operation happens in place. Note that this word is language specific and is written to handle English only.

```
: ERASE \ a-addr u --
```

Erase U bytes of memory from A-ADDR with 0.

```
: BLANK \ a-addr u --
```

Blank U bytes of memory from A-ADDR using ASCII 32 (space).

1.8 Dictionary management

```
: HERE \ -- addr
```

Return the current dictionary pointer which is the first address-unit of free space within the system.

```
: ALLOT \ n --
```

Allocate N address-units of data space from the current value of HERE and move the pointer.

```
: aligned \ addr -- addr'
```

Given an address pointer this word will return the next ALIGNED address subject to system wide alignment restrictions.

: `ALIGN \ --`
ALIGN dictionary pointer using the same rules as ALIGNED.

: `LATEST \ -- c-addr`
Return the address of the name field of the last definition.

: `SMUDGE \ --`
Toggle the SMUDGE bit of the latest definition.

: `, \ x --`
Place the CELL value X into the dictionary at HERE and increment the pointer.

: `W, \ w --`
Place the WORD value X into the dictionary at HERE and increment the pointer. This word is not present on 16 bit implementations.

: `C, \ char --`
Place the CHAR value into the dictionary at HERE and increment the pointer.

: `there \ -- addr`
Harvard targets only: Return the DATA space pointer.

: `allot-ram \ n --`
Harvard targets only: ALLOT DATA space.

: `c,(r) \ b --`
Harvard targets only: The equivalent of C, for DATA space.

: `,(r) \ n --`
Harvard targets only: The equivalent of , for DATA space.

: `N>LINK \ a-addr -- a-addr'`
Move a pointer from a NFA field to the Link Field.

: `LINK>N \ a-addr -- a-addr'`
The inverse of N>LINK.

: `>LINK \ a-addr -- a-addr'`
Move a pointer from an XT to the link field address.

: `LINK> \ a-addr -- a-addr'`
The inverse of >LINK.

: >VOC-LINK \ wid -- a-addr

Step from a wordlist identifier, wid, to the address of the field containg the address of the previously defined wordlist.

: >#THREADS \ wid -- a-addr ; for XC5 compatibility

Step from a wordlist identifier, wid, to the address of the field containg the number of threads in the wordlist.

: >THREADS \ wid -- a-addr

Step from a wordlist identifier, wid, to the address of the array containing the top NFA for each thread in the wordlist.

: >VOCNAME \ wid -- a-addr

Step from a wordlist identifier, wid, to the address of the field pointing to the vocabulary name field.

: FIND \ c-addr -- c-addr 0|xt 1|xt -1

Perform the "SEARCH-WORDLIST" operation on all wordlists within the current search order. This definition takes a counted string rather than a c-addr u pair. The counted string is returned as well as the 0 on failure.

: .NAME \ nfa --

The correct way to display a definition's name given an NFA. string for a word name, return the address of the dictionary name thread that will contain the name.

: makeheader \ c-addr len --

Given a word name as string in addr/len form, build a dictionary header for the word.

: $CREATE \ c-addr --

Perform the action of CREATE (below) but take the name from a counted string. OBSOLETE: replace by:

```
count makeheader docreate,
```

: CREATE \ --

Create a new definition in the dictionary. When the new definition is executed it will return the address of the definition BODY.

1.9 String compilation

: (C") \ -- c-addr

The run-time action for C" which returns the address of and steps over a counted string.

`: (S") \ -- c-addr u`

The run-time action for S" which returns the address and length of and steps over a string.

`: (ABORT") \ i*x x1 -- | i*x`

The run time action of ABORT".

`: (.") \ --`

The run-time action of .".

1.10 Pre-ANS Exception handlers

Before the ANS Forth standard, these words were the primary error handlers. They are provided for compatibility, but wherever possible, the use of CATCH and THROW will be found to be more flexible.

`: ABORT \ i*x -- ; R: j*x --`

Performs "-1 THROW". This is a compatibility word for earlier versions of the kernel. Unfortunately, the earlier versions gave problems when ABORT was used in interrupt service routines or tasks. The new definition is brutal but consistent.

`: ABORT" \ Comp: "ccc<quote>" -- ; Run: i*x x1 -- | i*x ; R: j*x -- | j*x`

If x1 is non-zero at run-time, store the address of the following counted string in USER variable 'ABORTTEXT, and perform "-2 THROW". The text interpreter in QUIT will (if reached) display the text.

`: (Error) \ n --`

The default action of ERROR. This definition has been removed from v6.2 onwards. See the section about the changes from v6.1 to v6.2.

`: ?ERROR \ flag n --`

If flag is true, perform "n ERROR", otherwise do nothing. This definition has been removed from v6.2 onwards. See the section about the changes from v6.1 to v6.2.

1.11 ANS words CATCH and THROW

CATCH and THROW form the basis of all Forth error handling. The following description of CATCH and THROW originates with Mitch Bradley and is taken from an ANS Forth standard draft.

CATCH and THROW provide a reliable mechanism for handling exceptions, without having to propagate exception flags through multiple levels of word

nesting. It is similar in spirit to the "non-local return" mechanisms of many other languages, such as C's setjmp() and longjmp(), and LISP's CATCH and THROW. In the Forth context, THROW may be described as a "multi-level EXIT", with CATCH marking a location to which a THROW may return.

Several similar Forth "multi-level EXIT" exception-handling schemes have been described and used in past years. It is not possible to implement such a scheme using only standard words (other than CATCH and THROW), because there is no portable way to "unwind" the return stack to a predetermined place.

THROW also provides a convenient implementation technique for the standard words ABORT and ABORT", allowing an application to define, through the use of CATCH, the behavior in the event of a system ABORT.

1.11.1 Example implementation

This sample implementation of CATCH and THROW uses the non-standard words described below. They or their equivalents are available in many systems. Other implementation strategies, including directly saving the value of DEPTH, are possible if such words are not available.

SP@ (– addr) returns the address corresponding to the top of data stack.

SP! (addr –) sets the stack pointer to addr, thus restoring the stack depth to the same depth that existed just before addr was acquired by executing SP@.

RP@ (– addr) returns the address corresponding to the top of return stack.

RP! (addr –) sets the return stack pointer to addr, thus restoring the return stack depth to the same depth that existed just before addr was acquired by executing RP@.

```
nnn USER HANDLER   0 HANDLER !  \ last exception handler
: CATCH  ( xt -- exception# | 0 ) \ return addr on stack
        SP@ >R  ( xt )  \ save data stack pointer
        HANDLER @ >R    ( xt )  \ and previous handler
        RP@ HANDLER !   ( xt )  \ set current handler
        EXECUTE ( )     \ execute returns if no THROW
        R> HANDLER !    ( )     \ restore previous handler
        R> DROP ( )     \ discard saved stack ptr
        0       ( 0 )   \ normal completion
;
: THROW  ( ??? exception# -- ??? exception# )
        ?DUP IF ( exc# )        \ 0 THROW is no-op
                HANDLER @ RP!   ( exc# )        \ restore prev return stack
                R> HANDLER !    ( exc# )        \ restore prev handler
                R> SWAP >R      ( saved-sp ) \ exc# on return stack
                SP! DROP R>     ( exc# )        \ restore stack
                \       Return to the caller of CATCH because return
```

```
                \       stack is restored to the state that existed
                \       when CATCH began execution
        THEN
;
```

The ROM PowerForth implementation is similar to the one described above, but not identical.

1.11.2 Example use

If THROW is executed with a non zero argument, the effect is as if the corresponding CATCH had returned it. In that case, the stack depth is the same as it was just before CATCH began execution. The values of the i*x stack arguments could have been modified arbitrarily during the execution of xt. In general, nothing useful may be done with those stack items, but since their number is known (because the stack depth is deterministic), the application may DROP them to return to a predictable stack state.

Typical use:

```
: could-fail    \ -- char
  KEY DUP [CHAR] Q =
  IF  1 THROW  THEN
;

: do-it         \ a b -- c
  2DROP could-fail
;

: try-it                \ --
  1 2 ['] do-it CATCH IF
    ( -- x1 x2 ) 2DROP ." There was an exception" CR
  ELSE
    ." The character was " EMIT CR
  THEN
;

: retry-it              \ --
  BEGIN
    1 2 ['] do-it CATCH
   WHILE
    ( -- x1 x2 ) 2DROP  ." Exception, keep trying" CR
  REPEAT ( char )
  ." The character was " EMIT CR
;
```

1.11.3 Gotchas

If a THROW is performed without a CATCH in place, the system will/may crash. As the current exception frame is pointed to by the USER variable

HANDLER, each task and interrupt handler will need a CATCH if THROW is used inside it.

You can no longer use ABORT as a way of resetting the data stack and calling QUIT. ABORT is now defined as "-1 THROW".

: `CATCH \ i*x xt -- j*x 0|i*x n`

Execute the code at XT with an exception frame protecting it. CATCH returns a 0 if no error has occurred, otherwise it returns the throw-code passed to the last THROW.

: `THROW \ k*x n -- k*x|i*x n`

Throw a non-zero exception code n back to the last CATCH call. If n is 0, no action is taken except to DROP n.

: `?throw \ flag throw-code -- ; SFP017`

Perform a THROW of value throw-code if flag is non-zero, otherwise do nothing except discard flag and throw-code.

1.12 Formatted and unformatted i/o

1.12.1 Setting number bases

: `HEX \ --`

Change current radix to base 16.

: `DECIMAL \ --`

Change current radix to base 10.

: `OCTAL \ --`

Change current radix to base 8. 32 bit targets only.

: `BINARY \ --`

Change current radix to base 2.

1.12.2 Numeric output

: `HOLD \ char --`

Insert the ascii 'char' value into the pictured numeric output string currently being assembled.

: `SIGN \ n --`

Insert the ascii 'minus' symbol into the numeric output string if 'n' is negative.

: `# \ ud1 -- ud2`

Given a double number on the stack this will add the next digit to the pictured numeric output buffer and return the next double number to work with. PLEASE NOTE THAT THE NUMERIC OP STRING IS BUILT FROM RIGHT(lsd) to LEFT(msd).

: `#S \ ud1 -- ud2`

Keep performing # until all digits are generated.

: `<# \ --`

Begin definition of a new numeric output string buffer.

: `#> \ xd -- c-addr u`

Terminate defnition of a numeric output string. Returns address and length of the ascii result.

: `-TRAILING \ c-addr u1 -- c-addr u2`

Modify a string address/length pair to ignore any trailing spaces.

: `D.R \ d n --`

Output the double number 'd' using current radix, right justified to 'n' characters. Padding is inserted using spaces on the left side.

: `D. \ d --`

Output the double number 'd' without padding.

: `. \ n --`

Output the cell signed value 'n' without justification.

: `U. \ u --`

As with . but treat as unsigned.

: `U.R \ u n --`

As with D.R but uses a single-unsigned cell value.

: `.R \ n1 n2 --`

As with D.R but uses a single-signed cell value.

1.12.3 Numeric input

: `SKIP-SIGN \ addr1 len1 -- addr2 len2 t/f ; true if sign=negative`

Inspect the first character of the string, if it is a '+' or '-' character, step over the string. Returning true if the character was a '-', otherwise return false.

: `+DIGIT \ d1 n -- d2 ; accumulates digit into double accumulator`

Multiply d1 by the current radix and add n to it.

: `+CHAR \ char -- flag ; true if ok`

This routine handles non-numeric characters, returning true for valid characters. By default, the only acceptable non-numeric character is the double-number separator ','.

: `+ASCII-DIGIT \ d1 char -- d2 flag ; true=ok`

Accumulate the double number d1 with the conversion of char, returning true if the character is a valid digit or part of an integer.

: `(INTEGER?) \ c-addr u -- d/n/- 2/1/0`

The guts of INTEGER? but without the base override handling. See INTEGER?

: `Check-Prefix \ addr len -- addr' len'`

If any BASE override prefices or suffices are used in the input string, set BASE accordingly and return the string without the override characters.

: `Integer? \ $addr -- value type | 0`

Attempt to convert the counted string at 'addr' to an integer. The return result is either Zero for failed, One for a single-cell return result (followed by that cell) or Two for a double return. The ascii number string supplied can also contain implicit radix over-rides. A leading $ enforces hexadecimal, a leading # enforces decimal and a leading % enforces binary. The prefix '@' is supported for octal numbers in 32 bit systems, for which hexadecimal numbers can also be specified by a leading '0x' or a trailing 'h'.

: `>NUMBER \ ud1 c-addr1 u1 -- ud2 c-addr2 u2 ; convert all until non-digits`

Accumulate digits from string c-addr1/u2 into double number ud1 to produce ud2 until the first non-convertible character is found. c-addr2/u2 represents the remaining string with c-addr2 pointing the non-convertible character. The number base for conversion is defined by the contents of USER variable BASE.

1.13 String input and output

: `BS \ -- ; destructive backspace`

Perform a destructive backspace by issuing ASCII characters 8, 20h, 8. If OUT is non-zero at the start, it is decremented by one regardless of the actions of the device driver.

`: ?BS \ pos -- pos' step ; perform BS if pos non-zero`

If pos is non-zero and ECHOING is set, perform BS and return

`: ?EMIT \ char -- ; emit if echoing enabled`

If ECHOING is set, EMIT the character, otherwise discard it.

`: SAVE-CH \ char addr -- ; save as required`

Save char at addr, and output the character if ECHOING is set.

`: ." \ "ccc<quote>" --`

Output the text upto the closing double-quotes character.

`: $. \ c-addr -- ; display counted string`

Output a counted-string to the output device. Note that on Harvard targets (e.g. 8051) c-addr is in DATA space.

`: ACCEPT \ c-addr +n1 -- +n2 ; read up to LEN chars into ADDR`

Read a string of maximum size n1 characters to the buffer at c-addr, returning n2 the number of characters actually read. Input may be terminated by CR. The action may be input device specific. If ECHOING is non-zero, characters are echoed. If XON/XOFF is non-zero, an XON character is sent at the start and an XOFF character is sent at the the end.

1.14 Source input control

`0 value SOURCE-ID \ -- n ; indicates input source`

Returns an indicator of which device is generating source input. See the ANS specification for more details.

`: TIB \ -- c-addr ; return address of terminal i/p buffer`

Returns the address of the terminal input buffer. Note that tasks requiring user input must initialise the USER variable 'TIB. New code should use SOURCE and TO-SOURCE instead for ANS Forth compatibility.

`: TO-SOURCE \ c-addr u --`

Set the address and length of the system terminal input buffer. These are held in the user variables 'TIB and #TIB.

`: SOURCE \ -- c-addr u ; returns address and length of input source buffer`

Returns the address and length of the current terminal input buffer.

`: SAVE-INPUT \ -- xn..x1 n`

Save all the details of the input source onto the data stack. If it later becomes necessary to discard the saved input, NDROP will do the job. If you want to move the data to the return stack, N>R and NR> are available in some 32 bit implementations.

`: RESTORE-INPUT \ xn..x1 n -- flag`

Attempt to restore input specification from the data stack. If the stack picture between SAVE-INPUT and RESTORE-INPUT is not balanced, a non-zero is returned in place of N. On success a 0 is returned.

`: QUERY \ -- ; fetch line into TIB`

Reset the input source specification to the console and accept a line of text into the input buffer.

`: REFILL \ -- flag ; refill input source`

Attempt to refill the terminal input buffer from the current source. This may be a file or the console. An attempt to refill when the input source is a string will fail. The return result is a flag indicating success with TRUE and failure with FALSE. A failure to refill when the input source is a text file indicates the end of file condition.

1.15 Text scanning

`: PARSE \ char "ccc<char>" -- c-addr u`

Parse the next token from the terminal input buffer using <char> as the delimiter. The next token is returned as a c-addr u string description. Note that PARSE does not skip leading delimiters. If you need to skip leading delimiters, use PARSE-WORD instead.

`: PARSE-WORD \ char -- c-addr u ; find token in input stream, skip leading chars`

An alternative to WORD below. The return is a c-addr u pair rather than a counted string and no copy has occured, i.e. the contents of HERE are unaffected. Because no intermediate global buffers are used PARSE-WORD is more reliable than WORD for text scanning in multi-threaded applications and in winprocs.

`: WORD \ char "<chars>ccc<char>" -- c-addr`

Similar behaviour to the ANS PARSE definition but the returned string is described as a counted string.

1.16 Miscellaneous

`: HALT? \ -- flag`

Used in listed displays. This word will check the keyboard for a 'pause' key <space>, if the key is pressed it will then wait for a continue key or an abort key. The return flag is TRUE if abort is requested. Line Feed (LF, ASCII 10) characters are ignored.

`: origin- \ addr -- addr' ; normalise NFA to base of primary CDATA section`

If addr is non-zero, subtract the start address of the first defined CDATA section. This word is only compiled if the start address of the first defined CDATA section is non-zero.

`: origin+ \ addr -- addr' ; denormalise NFA again`

If addr is non-zero, add the start address of the first defined CDATA section. This word is only compiled if the start address of the first defined CDATA section is non-zero.

`: nfa-buff \ -- addr+len addr ; make a buffer for holding NFAs`

Form a temporary buffer for holding NFAs. A factor for WORDS.

`: MAX-NFA \ -- addr c-addr ; returns addr and top nfa`

Return the thread address and NFA of the highest word in the NFA buffer. A factor for WORDS.

`: COPY-THREADS \ addr --`

Copy the threads of the CONTEXT wordlist to a temporary NFA buffer for manipulation. A factor for WORDS.

`: WORDS \ --`

Display the names of all definitions in the wordlist at the top of the search-order.

`: MOVE \ addr1 addr2 u -- ; intelligent move`

An intelligent memory move, chooses between CMOVE and CMOVE> at runtime to avoid memory overlap problems. Note that as ROM PowerForth characters are 8 bit, there is an implicit connection between a byte and a character.

`: DEPTH \ -- +n`

Return the number of items on the data stack, excluding the count.

`: UNUSED \ -- u ; free dictionary space`

Return the number of bytes free in the dictionary.

`: .FREE \ --`

Return the free dictionary space.

1.17 Wordlist control

: `WORDLIST \ -- wid`

Create a new wordlist and return a unique identifier for it.

: `VOCABULARY \ -- ; VOCABULARY <name>`

Create a VOCABULARY which is implemented as a named wordlist.

: `FORTH \ --`

Install FORTH wordlist into search-order.

: `FORTH-WORDLIST \ -- wid`

Return the unique WID for the main FORTH wordlist.

: `GET-CURRENT \ -- wid`

Return the WID for the Wordlist which holds any definitions made at this point.

: `SET-CURRENT \ wid --`

Change the wordlist which will hold future definitions.

: `GET-ORDER \ -- widn...wid1 n`

Return the list of WIDs which make up the current search-order. The last value returned on top-of-stack is the number of WIDs returned.

: `SET-ORDER \ widn...wid1 n -- ; unless n = -1`

Set the new search-order. N is the number of WIDs to place in the search-order. If N is -1 then the minimum search order is inserted.

: `ONLY \ --`

Set the minimum search order as the current search-order.

: `ALSO \ --`

Duplicate the first WID in the search order.

: `PREVIOUS \ --`

Drop the current top of search-order.

: `DEFINITIONS \ --`

Set the current top WID of search-order as the current definitions wordlist.

1.18 Control structures

`: ?PAIRS \ x1 x2 --`

If x1<>x2, issue and error. Used for on-target compile-time error checking.

`: !CSP \ x --`

Save the stack pointer in CSP. Used for on-target compile-time error checking.

`: ?CSP \ --`

Issue an error if the stack pointer is not the same as the value previously stored in CSP. Used for on-target compile-time error checking.

`: ?COMP \ --`

Error if not in compile state.

`: ?EXEC \ --`

Error if not interpreting.

`: DO \ C: -- do-sys ; Run: n1|u1 n2|u2 -- ; R: -- loop-sys`

Begin a DO ... LOOP construct. Takes the end-value and start-value from the data-stack.

`: ?DO \ C: -- do-sys ; Run: n1|u1 n2|u2 -- ; R: -- | loop-sys`

Compile a DO which will only begin loop execution if the loop parameters are not the same. Thus 0 0 ?DO ... LOOP will not execute the contents of the loop.

`: LOOP \ C: do-sys -- ; Run: -- ; R: loop-sys1 -- | loop-sys2`

The closing statement of a DO..LOOP construct. Increments the index and terminates when the index crosses the limit.

`: +LOOP \ C: do-sys -- ; Run: n -- ; R: loop-sys1 -- | loop-sys2`

As with LOOP except you specify the increment on the data-stack.

`: BEGIN \ C: -- dest ; Run: --`

Mark the start of a BEGIN..[while]..UNTIL / AGAIN / [REPEAT] construct.

`: AGAIN \ C: dest -- ; Run: --`

The end of a BEGIN..AGAIN construct which specifies an infinite loop.

: `UNTIL \ C: dest -- ; Run: x --`

Compile code into definition which will jump back to the matching BEGIN if the supplied condition flag is Zero / FALSE.

: `WHILE \ C: dest -- orig dest ; Run: x --`

Separate the condition test from the loop code in a BEGIN..WHILE..REPEAT block.

: `REPEAT \ C: orig dest -- ; Run: --`

Loop back to the conditional dest code in a BEGIN..WHILE..REPEAT construct.

: `IF \ C: -- orig ; Run: x --`

Mark the start of an IF..[ELSE]..THEN conditional block.

: `THEN \ C: orig -- ; Run: --`

Mark the end of an IF..THEN or ..ELSE..THEN conditional.

: `endif \ C: orig -- ; Ru: -- ; synonym for THEN`

An alias for THEN. Note that ANS Forth describes THEN not ENDIF.

: `AHEAD \ C: -- orig ; Run: --`

Start an unconditional forward branch which will be resolved later.

: `ELSE \ C: orig1 -- orig2 ; Run: --`

Begin the failure condition code for an IF.

: `CASE \ C: -- case-sys ; Run: --`

Begin a CASE..ENDCASE construct. Similar to 'C' language switch.

: `OF \ C: -- of-sys ; Run: x1 x2 -- | x1`

Begin conditional block for CASE, executed when the switch value is equal to the X2 value placed in TOS.

: `ENDOF \ C: case-sys1 of-sys -- case-sys2 ; Run: --`

Mark the end of an OF conditional block within a CASE construct. Compile a jump past the ENDCASE marker at the end of the construct.

: `ENDCASE \ C: case-sys -- ; Run: x --`

Terminate a CASE..ENDCASE construct. DROPs the switch value off the stack.

: `?OF \ C: -- of-sys ; Run: flag --`

Begin conditional block for CASE, executed when the flag is true.

: END-CASE \ C: case-sys -- ; Run: --

A Version of ENDCASE which does not drop the switch value. Used when the switch value itself is consumed by a DEFAULT condition.

: NEXTCASE \ C: case-sys -- ; Run: x --

Terminate a CASE..NEXTCASE construct. DROPs the switch value from the stack and compiles a branch back to the top of the loop at CASE.

: RECURSE \ Comp: --

Compile a recursive call to the colon definition containing RECURSE itself. Do not use RECURSE between DOES> and ;. Used in the form:

```
: foo ... recurse ... ;
```

to compile a reference to FOO from inside FOO.

1.19 Target interpreter and compiler

: ?STACK \ --

Error if stack pointer out of range.

: ?UNDEF \ x --

Word not defined error if x=0.

: (compile) \ -- ; compiles in line xt

The run-time action for COMPILE and friends.

: POSTPONE \ Comp: "<spaces>name" --

Compile a reference to another word. POSTPONE can handle compilation of IMMEDIATE words which would otherwise be executed during compilation.

: S" \ Comp: "ccc<quote>" -- ; Run: -- c-addr u

Describe a string. Text is taken upto the next double-quote character. The address and length of the string are returned.

: C" \ Comp: "ccc<quote>" -- ; Run: -- c-addr

As with S" except the address of a counted string is returned.

: #LITERAL \ n1 .... nn n -- ; put in dictionary n1 first

Compile n1..nn as literals so that the same stack order results when the code executes.

: LITERAL \ Comp: x -- ; Run: -- x

Compile a literal into the current definition. Usually used in the form [<expression] LITERAL inside a colon definition. Note that LITERAL is IMMEDIATE.

: 2LITERAL \ Comp: x1 x2 -- ; Run: -- x1 x2

A two cell version of LITERAL.

: CHAR \ "<spaces>name" -- char

Return the first character of the next token in the input stream. Usually used to avoid magic numbers in the source code.

: [CHAR] \ Comp: "<spaces>name" -- ; Run: -- char

Compile the first character of the next token in the input stream as a literal. Usually used to avoid magic numbers in the source code.

: sliteral \ c-addr u -- ; Run: -- c-addr2 u ; 17.6.1.2212

Compile the string c-addr1/u into the dictionary so that at run time the identical string c-addr2/u is returned. Note that because of the use of dynamic strings at compile time the address c-addr2 is unlikely to be the same as c-addr1.

: [\ --

Switch compiler into interpreter state.

:] \ --

Switch compiler into compilation state.

: IMMEDIATE \ --

Mark the last defined word as IMMEDIATE. Immediate words will execute whenever encountered regardless of STATE.

: ' \ "<spaces>name" -- xt

Find the xt of the next word in the input stream. An error occurs if the xt cannot be found.

: ['] \ Comp: "<spaces>name" -- ; Run: -- xt

Find the xt of the next word in the input stream, and compile it as a literal. An error occurs if the xt cannot be found.

: [COMPILE] \ "<spaces>name" --

Compile the next word in the input stream. [COMPILE] ignores the IMMEDIATE state of the word. Its operation is mostly superceded by POSTPONE.

: (\ "ccc<paren>" --

Begin an inline comment. All text upto the closing bracket is ignored.

: \ \ "ccc<eol>" --

Begin a single-line comment. All text up to the end of the line is ignored.

: ", \ "ccc<quote>" --

Parse text up to the closing quote and compile into the dictionary at HERE as a counted string. The end of the string is aligned.

: .(\ "cc<paren>" --

A documenting comment. Behaves in the same manner as (except that the enclosed text is written to the console at compile time.

: ASSIGN \ "<spaces>name" --

A state smart word to get the XT of a word. The source word is parsed from the input stream. Used as part of a ASSIGN xxx TO-DO yyy construct.

: (TO-DO) \ -- ; R: xt -- a-addr'

The run-time action of TO-DO. It is followed by the data addres of the DEFERred word at which the xt is stored.

: TO-DO \ "<spaces>name" --

The second part of the ASSIGN xxx TO-DO yyy construct. This word will assign the given XT to be the action of a DEFERed word which is named in the input stream.

: exit \ R: nest-sys -- ; exit current definition

Compile code into the current definition to cause a definition to terminate. This is the Forth equivalent to inserting an RTS/RET instruction in the middle of an assembler subroutine.

: ; \ C: colon-sys -- ; Run: -- ; R: nest-sys --

Complete the definition of a new 'colon' word or :NONAME code block.

: INTERPRET \ --

Process the current input line as if it is text entered at the keyboard.

: N>R \ xn .. x1 N -- ; R: -- x1 .. xn n

Transfer N items and count to the return stack.

: NR> \ -- xn .. x1 N ; R: x1 .. xn N --

Pull N items and count off the return stack.

`: EVALUATE \ i*x c-addr u -- j*x ; interpret the string`

Process the supplied string as though it had been entered via the interpreter.

`: .throw \ throw# --`

Display the throw code. Values of 0 and -1 are ignored.

`: QUIT \ -- ; R: i*x --`

Empty the return stack, store 0 in SOURCE-ID, and enter interpretation state. QUIT repeatedly ACCEPTs a line of input and INTERPRETs it, with a prompt if interpreting and ECHOING on. Note that any task that uses QUIT must initialise 'TIB, BASE, IPVEC, and OPVEC.

1.20 Compilation and Caches

Because some CPUs, e.g. StrongARM, have separate instruction and data caches, self-modifying code can cause problems when code is laid down (into the Dcache) and then an attempt is made to execute it (the Icache will not necessarily contain the code). For this reason a word is provided that will synchronise the caches for an address range. This word is CPU specific and may reference code in a CPU and/or hardware specific file.

Synchronisation will usually only be necessary when creating words, constants, variables etc. interactively on the target and then executing them before the code has got into the Icache. Only executable code has to be synchronised, not data.

If the word FLUSHCACHE (–) is provided before KERNEL62.FTH is compiled, it will be executed by the text interpreter before each line is processed. FLUSHCACHE is also executed by ';'.

1.21 Startup code

1.21.1 Cold chain

If enabled by the non-zero equate COLDCHAIN? the cold start code in COLD will walk a list and execute the xts contained in it. The xts must have no stack effect (–) and are added to the list by the phrase:

`' <wordname> AtCold`

The list is executed in the order in which it was defined so that the last word added is executed last. This was done for compatibility with VFX Forth, which also contains a shutdown chain, in which the last word added is executed first.

If the equate COLDCHAIN? is not defined in the control file, a default value of 0 will be defined.

`l: ColdChainFirst \ -- addr`

Dummy first entry in ColdChain.

`variable ColdChain \ -- addr`

Holds the address of the last entry in the cold chain.

`: AtCold \ xt --`

Specifiy a new XT to execute when COLD is run. Note that the last word added is executed last. ATCOLD can be executed interpretively during cross-compilation. The cold chain is built in the current CDATA section.

`: WalkColdChain \ -- MPE.0000`

Execute all words added to the cold chain. Note that the first word added is executed first.

1.21.2 The COLD sequence

At power up, the target executes COLD or the word specified by MAKE-TURNKEY <name>.

`: (INIT) \ --`

Performs the high level Forth startup. See the source code for more details.

`: COLD \ --`

The first high level word executed by default. This word is set to be the word executed at power up, but this may be overridden by a later use of MAKE-TURNKEY <name>. See the source code for more details of COLD.

1.22 Kernel error codes

-4 Data stack underflow.

-13 Undefined word.

-14 Attempt to interpret a compile only definition.

-22 Control structure mismatch - unbalanced control structure.

-121 Attempt to remove with MARKER or FORGET below FENCE in protected dictionary.

-403 Attempt to compile an interpret only definition.

-501 Error if not LOADing from a block.

1.23 Differences between the v6.1 and 6.2 kernels

1.23.1 Error handling

All error handling in the v6.2 kernel is defined in terms of CATCH and THROW. The earlier words ERROR and ?ERROR have been removed. If you need them, define then as synonyms for THROW and ?THROW.

The definition of ABORT has changed significantly. The old version was:

```
: ABORT         \ i*x -- ; R: j*x --
\ *G Empty the data stack and perform the action of QUIT, which includes
\ ** emptying the return stack, without displaying a message.
  xon/xoff off  echoing on              \ No Xon/Xoff, do Echo
  s0 @ sp!                              \ reset data stack
  quit                                  \ start text interpreter
;
```

The new version is:

```
: ABORT         \ i*x -- ; R: j*x --
\ *G Performs "-1 THROW". This is a compatibility word for earlier
\ ** versions of the kernel. Unfortunately, the earlier versions
\ ** gave problems when ABORT was used in interrupt service routines
\ ** or tasks. The new definition is brutal but consistent.
  -1 Throw
;
```

The old version worked 99% of the time, except that in tasks or interrupt service routines, the result was unpredictable. Because modern applications are larger and more complex, ABORT has to be completely predictable. The line

```
xon/xoff off echoing on \ No Xon/Xoff, do Echo
```

is now part of QUIT. The phrase "S0 @ SP!" must now be provided by the THROW handler.

The previous definition of THROW checked for a previously defined CATCH and performed the old ABORT if no CATCH had been defined. The new version assumes that a CATCH has been defined and may/will crash if no CATCH has been performed. The result is a faster and smaller definition of CATCH. However, it is now the programmer's responsibility to provide a CATCH handler for ALL ISRs and tasks that may generate a THROW. This is actually very little different from the previous situation, except that the system is less forgiving if you forget to provide a handler.

Error codes have been made ANS compliant. It is MPE policy that all error and ior (i/o result) codes shall be distinct from now on.

1.23.2 Terminal input buffer and ACCEPT.

The changes below simplifiy the source code, and permit multiple tasks to use EVALUATE without interaction. Note that compilation from multiple sources/tasks requires the interpreter/compiler to interlocked with a semaphore.

The 2VARIABLE SOURCE-STRING has been removed, and TO-SOURCE and SOURCE use 'TIB and #TIB instead.

The state variables ECHOING and XON/XOFF are now USER variables. In most cases this will have no impact. However, tasks may now control these variables independently.

QUIT always enforces ECHOING on and disables XON/XOFF processing. QUIT does not select an I/O device. This change was made to allow the interpreter to be used on any channel in systems with several serial lines or with the Telnet service of the PowerNet TCP/IP stack. Note that any task that uses QUIT must initialise IPVEC, OPVEC, ECHOING and XON/XOFF.

Removed: ?EMIT SOURCE-STRING

Chapter 2

Character Queues

The file COMMON\CQUEUES.FTH provides circular character (byte) queues. If the equate TASKING? is non-zero, the blocking routines will use PAUSE. Interrupts are disabled for the queue empty/full checks.

`struct /cqueue \ -- size ; character queue structure in idata, buffer in udata`

Circular queue data structure.

```
   int >qhead            \ Offset of head
  int >qtail            \ Offset of tail
  int >qchars           \ Number of characters in the queue
  int >qmask            \ Mask to apply to pointers
  ptr >qbuffer          \ Base address of character buffer
end-struct
```

`: cqueue: \ size -- ; -- cqueue ; size CQUEUE: <name>`

An interpreter definition to build a character queue of the specified size. The queue data structure is built in the current IDATA space, and the buffer itself is in the current UDATA space. Executing <name> returns the address of the queue data structure. N.B. The size of a queue must be a power of two, e.g. 32, 64 ...

`: init-cqueue \ cqueue -- ; initialise queue`

Initialise the specified queue created by CQUEUE:.

`: init-hcqueue \ size cqueue -- ; SFP002`

Initialise the specified queue created by ALLOCATE. When a queue is allocated from the heap by a phrase of the form "/CQUEUE <size> + ALLOCATE", this word must be used.

`: (>cqueue) \ char cqueue -- ; put character on cqueue`

Put char into the queue with no checks.

: `(cqueue>) \ cqueue -- char ; get next character from queue`
Get the next character from the queue with no checks.

: `(cqfull?) \ queue -- flag ; TRUE if queue full`
Return true if the queue is full. No interrupt protection is provided.

: `cqfull? \ queue -- flag ; TRUE if queue full`
Return true if the queue is full. Interrupt protection is provided.

: `cqchars \ queue -- n`
Return the number of characters in the queue.

: `cqempty? \ queue -- flag ; TRUE if queue empty`
Return true if the queue is empty.

: `cqnotempty? \ queue -- flag ; TRUE if queue not empty`
Return true if the queue is not empty, i.e. if it contains any characters.

: `>cqueue \ char cqueue -- ; spins if full`
Put a character into the queue. If the queue is full, the system waits (blocks) until there is enough space.

: `cqueue> \ queue -- char ; spins while queue empty`
Remove the next character, waiting if the queue is empty.

Chapter 3

Heap Memory Allocation

3.1 Heap definition

The heap is allocated from a predefined section of memory. Facilities are provided for user expansion of the heap to mass storage, although the current code makes no provision for page management. When the heap is initialised, a free block and an end block are created. The end block is of zero size, and is used only as a marker. The address returned by ALLOCATE and RESIZE is the address of the first data byte, as is the address consumed by FREE.

The heap MUST be initialised before use by calling INIT-HEAP. Heap access words return status=0 for success, and status<>0 for error.

Two equates are required during compilation to allocate a contiguous block of RAM for the heap.

```
STARTOFHEAP is the start address of the heap
SIZEOFHEAP is the size of the RAM for the heap
```

There are two versions of this code provided. HEAP32.FTH is provided for 32 bit targets and is optimised for the VFX code generator. HEAP16.FTH is for 16 bit targets, and is optimised for code density.

3.1.1 16 bit targets - HEAP16.FTH

The heap is controlled using two cells per block. This information is used in three parts:

```
cell = #bytes, number of bytes in this block
cell = flag, split between a four bit and a 12 bit field
```

The top four bits of the flag are used to indicate the block type, where $E = End, $F = Free, $A = Allocated. Others may be added later for type management.

The bottom 12 bits of the flag are currently unused, and should be set to zero.

3.1.2 32 bit targets - HEAP32.FTH

The heap is controlled using a single cell per block. This information is used in two parts:

```
bits 31..24: $EE - End, $FF - Free, $AA - Allocated
bits 23..0: 24 bits for number of data bytes in block.
```

A consequence of this is that the maximum block size that can be allocated is 16Mb-1 bytes.

If you use a pre-emptive scheduler or need to use the heap routines inside interrupt routines, you must define suitable heap lock and unlock routines and set the equate LOCKHEAP? to non-zero.

LockHeap=0 no heap locking

LockHeap=1 heap locking by turning off interrupts

LockHeap=2 heap locking by semaphore.

3.2 Gotchas

The heap routines must be protected if they are to be used both in normal code and in interrupts. In this case the code must be modified to be interrupt safe, but this may have a significant impact on interrupt latency. Examples may be found in HEAP32.FTH.

3.3 Glossary

The glossary does not include all the factors used in the code. If you are interested in the implementation, please read the sources.

`: allocate \ #bytes -- addr status`

Attempt to allocate some memory from the heap. Walk the heap looking for a single big enough block. If the block is larger than than required split it into two blocks. Allocate part or all of the free block. Status=0 for success.

`: free \ address -- status`

Attempt to free a heap block. Status=0 for success.

`: resize \ addr1 size -- addr2 ior`

Try to resize an allocated block to a new size, allowing for alignment. If the existing memory block is not big enough, the data will be copied to a new block, and the returned addr2 will not be the same as addr1. Status=0 for success.

`: init-heap \ -- ; initialise the heap structures`

The heap is initialised by creating 2 blocks. Block 1 starts at the beginning and is marked as a free block. Block 2 Is a null marker at the end of heap space.

3.4 Diagnostics

`: size \ addr -- currsize | -1`

Return the size of an allocated block or -1 if there's an error.

`: .heap \ -- ; display heap info`

Walk the heap displaying block information.

`: heapok? \ -- t/f ; check heap`

Walk the heap and return TRUE if the heap is "well".

Chapter 4

Target VALUE and local variables

The file COMMON\METHODS.FTH implements the compilation of VALUEs and the ANS Forth LOCALS— syntax for compilation on the target. Compilation of this file requires CPU dependent support, usually called LOCAL.FTH in the %CpuDir% directory, and MPE standard control files will compile these files if the equate TARGET-LOCALS? is set non-zero in the control file.

Note that this file is only provided for full ANS compliance. The MPE extended local variable syntax is provided by the cross compiler, and is much more powerful and more readable.

Note also that compilation of %CpuDir%\LOCAL.FTH may be required if you cross compile words with more than four input arguments.

`: OPERATOR \ n -- ; define an operator in the cross compiler`

An interpreter definition that build new operators such as "to" and "addr".

`: VALUE \ n -- ; -- n ; n VALUE <name>`

Creates a variable of initial value n that returns its contents when referenced. To store to a child of VALUE use "n to <child>".

`: (LOCAL) \ Comp: c-addr u -- ; Exec: -- x ; define local var`

When executed during compilation, defines a local variable whose name is given by c-addr/u. If u is zero, c-addr is ignored and compilation of local variables is assumed to finish. When the word containing the local variable executes, the local variable is initialised from the stack. When the local variable executes, its value is returned. The local variable may be written to by preceding its name with TO. This word is provided for the construction of user-defined local variable notations. This word is only provided for ANS compatibility, and locals created by it cannot be optimised by the VFX code generator.

`: LOCALS| \ "name...name |" --`

Create named local variables <name1> to <namen>. At run time the stack effect is (xn..x1 –), such that <name1> is initialised with x1 and <namen> is initialised with xn. Note that this means that the order of declaration is the reverse of the order used in stack comments! When referenced, a local variable returns its value. To write to a local, precede its name with TO.

In the example below, a and b are named inputs.

```
: foo      \ a b --
  locals| b a |
  a b +  cr .
  a b *  cr .
;
```

Chapter 5

Target local variables

The file COMMON\LOCALCOM.FTH implements the compilation of the ANS Forth LOCALS— syntax for compilation on the host. Compilation of this file requires CPU dependent support, usually called LOCAL.FTH in the %CpuDir% directory, and MPE standard control files will compile these files if the equate TARGET-LOCALS? is set non-zero in the control file.

Note that this file is only provided for full ANS compliance. The MPE extended local variable syntax is provided by the cross compiler, and is much more powerful and more readable.

`:  (LOCAL) \ Comp:  c-addr u -- ; Exec:  -- x ; define local var`

When executed during compilation, defines a local variable whose name is given by c-addr/u. If u is zero, c-addr is ignored and compilation of local variables is assumed to finish. When the word containing the local variable executes, the local variable is initialised from the stack. When the local variable executes, its value is returned. The local variable may be written to by preceding its name with TO. This word is provided for the construction of user-defined local variable notations. This word is only provided for ANS compatibility, and locals created by it cannot be optimised by the VFX code generator.

`:  LOCALS| \ "name...name |" --`

Create named local variables <name1> to <namen>. At run time the stack effect is (xn..x1 –), such that <name1> is initialised with x1 and <namen> is initialised with xn. Note that this means that the order of declaration is the reverse of the order used in stack comments! When referenced, a local variable returns its value. To write to a local, precede its name with TO.

In the example below, a and b are named inputs.

```
: foo      \ a b --
  locals| b a |
```

```
  a b +  cr .
  a b *  cr .
;
```

Chapter 6

Ethernet and IP devices generic I/O

6.1 Internet Protocol Devices

In order to ease changing devices and to permit systems with multiple ports, version 3 and above of the Powernet TCP/IP stack use a generalised interface to the hardware, which is usually an Ethernet driver or a serial driver.

The interface consists of a vector (table) of XTs corresponding to the particular function required. The layout of the table is as follows:

```
create IPdevice
  ' MyInit ,       \ 0: initialisation
  ' MyTerm ,    \ 1: shutdown
  ' MyRx? ,     \ 2: receive test
  ' MyRx                \ 3: receive packet
  ' MyTx? ,     \ 4: transmit test
  ' MyTx ,              \ 5: transmit packet
  ' MyGetAddr ,    \ 6: Get device addresses
  ' MySetAddr ,    \ 7: Set device addresses
  ' MySave ,    \ 8: Save IP device addresses and state
  ' MyDiscard , \ 9: Discard current receive packet
```

by default, the system code requires a default IP device called IPDevice. IPDevice will be used when PowerNet is started.

`cell +User IPDVec \ -- addr`

IPDVec is a USER variable which contains the address of the current Internet Protocol Device (IPD) vector. All tasks must initialise IPDVEC before using the ETHERCOM interface.

```
IPDevice IPDVec !
```

`: IPDinit \ --`

Initialise (open) the current IP device.

`: IPDterm \ --`

Shutdown (close) the current IP device.

`: IPDRx? \ -- flag ; check receive char`

Return true if a packet is available at the current IP device.

`: IPDRx \ buff size -- len ; receive packet`

Receive a packet into buff of size bytes, returning len the number of bytes received. Use IPDRX? to check that a packet is available.

`; IPDTx? \ len -- flag ; check TX capability`

Return true if the current IP device can send a packet of size len bytes.

`: IPDTx \ buff len -- ; send packet`

Transmit the given packet.

`: IPDGetAddr \ -- ipaddr netaddr flags`

Returns: a pointer to the IP address used by this device, or 0 if it has not been set yet: a pointer to the network address of the device, e.g. the Ethernet MAC address or 0 if not set or irrelevant: and a set of flags which are currently 0 for IPv4 and Ethernet/SLIP/PPP devices.

`: IPDSetAddr \ ipaddr netaddr flags --`

Consumes: a pointer to the IP address used by this device, or 0 for no change: a pointer to the network address of the device, e.g. the Ethernet MAC address or 0 for no change: and a set of flags which are currently 0 for IPv4 and Ethernet/SLIP/PPP devices.

`: IPDSave \ --`

Save the device state to non-volatile storage so that it can be reloaded at the next power up or IPDinit.

`: IPDDiscard \ --`

Discard the current receive packet, usually because PowerNet has run out of buffer space. This is only valid if IPDRX? has returned true to say that a packet is available.

Chapter 7

Development tools

The file COMMON\DEVTOOLS.FTH supplies words that are most used during development and debugging.

`1 equ simple?  \ -- n`

Set this flag non-zero to generate .xWORD to avoid divisions. On some CPUs, a division operation is slow.

`: .nibble \ n --`

Convert a nibble to a hex ASCII digit and display it.

`: .BYTE \ b --`

Display b as two hex digits.

`: .WORD \ w --`

Display w as four hex digits.

`: .LWORD \ x --`

Display x as eight hex digits. The separator ":" makes the output easier to read. Future releases of MPE Forths will treat the ":" character as having no effect on number input parsing. This character is chosen because it does not conflict with the current use of the "." and "," characters for numbers. This word is only compiled for 32 bit targets.

`: .DWORD \ x --`

A synonym for .LWORD.

`: .ASCII \ char --`

The top bit of char is zeroed. If char is in the range 32..126 it is displayed, otherwise a "." is displayed.

`: DUMP \ addr len --`

Display (dump) len bytes of memory starting at addr.

`: LDUMP \ addr len -- ; dump 32 bit long words`

Display (dump) len bytes of memory starting at addr as 32 bit words.

`: WDUMP \ addr len -- ; dump 16 bit half words`

Display (dump) len bytes of memory starting at addr as 16 bit half-words.

`: .S \ --`

Display the contents of the data stack without affecting it.

`: ? \ a-addr --`

Display contents of a memory location as a cell.

Chapter 8

Debugging tools

MicroProcessor Engineering
133 Hill Lane
Southampton SO15 5AF
England

tel: +44 (0)23 8063 1441
fax: +44 (0)23 8033 9691
net: mpe@mpeltd.demon.co.uk
tech-support@mpeltd.demon.co.uk
web: www.mpeltd.demon.co.uk

The file *Common\DebugTools.fth* provides debugging tools for MPE embedded systems created by Forth 6 Cross Compilers. The emphasis is on 32 bit systems and interactive testing. The tools can easily be ported to other systems. Copyright is retained by MPE. The code may be freely used on non-MPE systems for non-commercial use. The copyright notice must be preserved.

Porting the code to other systems is up to you. This code may require some carnal knowledge of how your system works. Most Forths contain the required words, but they may not have the same names that MPE use.

8.1 Implementation dependencies

In MPE embedded systems, the `USER` variables `IPVEC` and `OPVEC` contain the address of the device structure used for input and output by `KEY`, `EMIT` and friends. In VFX Forth for Windows/Linux, the variables are `IP-HANDLE` and `OP-HANDLE`.

`: consoleIO \ --`

Select debug console for output. By default this is the `CONSOLE` device.

```
   console opvec !  console ipvec !
  Echoing on  Xon/Xoff off
;
```

```
:  name?  \ addr -- flag MPE.0000
```

Check to see if the supplied address is a valid NFA, returning true if the address appears to be a valid NFA. This word is implementation dependent. For MPE cross compilers, a valid NFA for MPE embedded systems satisfies the following:

- All characters within string are printable ASCII within range 33..126
- String Length is non-zero in range 1..31 and bit 7 is set, ignore bits 6, 5

```
   count                                  \ c-addr u --
  dup  $9F and  $81 $9F within? 0=        \ NFA first byte = 1SIxxxxx, count = xxxxx
                                          \ mask           = 10011111
  if  2drop  0  exit  then
  $01F and bounds ?do
    i c@ #33 #126 within? 0=              \ check all ascii chars
    if  unloop  FALSE exit  then
  loop
  TRUE
;
```

```
:  ip>nfa \ addr -- nfa
```

Attempt to move backwards from an address within a definition to the relevant NFA.

```
   2-                              \ NFA must be at least 'n' bytes backwards
  begin
    dup name? 0=
   while
    1-
  repeat
;
```

```
:  >name \ xt -- nfa
```

Move from a word's xt to its name field. If >NAME does not exist IP>NFA will be used.

```
   ip>nfa
;
```

```
:  .name \ nfa --
```
Given a word's NFA display its name.

```
   count $1F and type
;


:  .DWORD \ dw --
```
Display the 32 bit long word 'dw' as an 8 digit hex number.

```
   base @ hex  swap
  0 <# # # # # ascii : hold # # # # #> type
  base !
;
```

8.2 Miscellaneous

MPE systems use `TICKS ( -- ms)` to return a running time count in milliseconds. Windows systems can use the **GetTickCount** API call.

```
:  times \ n -- ; n TIMES <word>
```
Execute <word> n times, and display the execution time. The ticker interrupt must be running.

```
   ticks  '  rot 0                        \ -- ticks xt n 0
  ?do  dup execute  loop
  drop
  ticks swap - . ." ms"
;


:  .ColdChain \ --
```
Display all words added to the cold chain. Note that the first word added is displayed first. In VFX Forth this word is called ShowColdChain.

```
   cr  ColdChainFirst
  begin
    dup
   while
    dup cell + @ >name .name            \ execute XT
    @                                   \ get next entry
  repeat
  drop
;
```

```
:  .decimal \ n --
```

Display a value as a decimal number.

```
   base @ >r  decimal .  r> base !
;
```

```
:  .hex \ n --
```

Display a value as a hexadecimal number.

```
   base @ >r  hex u.  r> base !
;
```

```
:  [con \ -- ; R: -- consys
```

Saves `BASE` and the current i/o vectors on the return stack, and then switches to the console and decimal.

```
    r>
   base @ >r  opvec @ >r  ipvec @ >r
   ConsoleIO decimal
   >r
;
```

```
:  con] \ -- ; R: consys --
```

Restores `BASE` and the current i/o vectors from the return stack.

```
   r>
  r> ipvec !  r> opvec !  r> base !
  >r
;
```

```
:  CheckFailed \ ip caddr len --
```

Given the address at the fault occurred and a string, ouput the string and some diagnostic information.

```
   [con
    cr type  ."  failed at "
    dup .dword ."  in " ip>nfa .name
  con]
;
```

8.3 Stack checking

Especially in multi-tasked systems, stack errors can be fatal. Detecting them as early as possible reduces debugging time. These words rely on Forth return stack cells containing return addresses. This is true on the vast majority of Forth systems except for some 8051 and real-mode 80x86 systems. If you find others, please let us know.

```
:  ?StackDepth \ +n --
```

If the stack depth before +n is not n, issue a console warning message and clear the stack. Note that this word is implementation dependent.

```
   dup 2+ depth =
  if  drop  exit  endif                  \ no failure
  [con
  cr ." *** Stack fault: depth = " depth 1- 0 .r ." (d) "
  ." in task " self .task                \ indicate current task
  >r  s0 @ sp!  r> 0 ?do  0  loop        \ set required depth
  cr ."   Stack updated."
  con]
;


:  ?StackEmpty \ --
```

If the stack depth is non-zero, issue a console warning message and clear the stack.

```
   0 ?StackDepth
;


:  TaskChecks \ --
```

Use in task to check for creeping stacks and so on. This word can be extended to provide additional internal consistency checks.

```
   ?StackEmpty
;


:  SF{ \ n -- ; R: -- depth
```

n SF{ .... }SF will check for stack faults. n describes the stack change between SF{ and }SF. If the stack change is different, an error message is generated. This word will work on most systems in which the return address is held on the return stack.

```
   r>  swap depth 2- + >r  >r
;
```

```
:  }SF \ -- ; R: depth -- ; perform stack check
```

The end of an SF{ ... }SF structure. This word is not strictly portable as it assumes that the Forth return stack holds a valid return address. In the vast majority of cases the assumption is true, but beware of some 8051 implementations. See SF{

```
   r>
  r> depth 2- <> if
    dup s" Stack check" CheckFailed
  endif
  >r
;
```

8.4 Assertions

Assertions are a useful way to check that the system is behaving correctly. When the phrase:

```
 [ASSERT <test> ASSERT]
```

is compiled into a piece of code, the test is performed and generates an error report if the result is false. If you do not want the performance overhead of the test, set the value `ASSERTS?` to zero. To remove even the small overhead of of testing `ASSERTS?`, comment out the line.

```
-1 value assert?  \ -- n
```

Returns non-zero if asserts will be tested.

```
:  (assert) \ flag --
```

If flag is zero, report an ASSERT error.

```
   if  exit  endif                          \ faster on some CPUs
  r@ s" ASSERT" CheckFailed
;
```

```
:  [assert \ --
```

Compile the code to start an assert.

```
   ?comp                                    \ must be compiling
  postpone assert?  postpone if
; immediate
```

```
:  assert] \ --
```

Compile the code to end an assert.

```
   ?comp                                \ must be compiling
  postpone (assert)  postpone then
; immediate
```

Here is a simple assert that will fail if `BASE` is not `DECIMAL`.

```
: foo     \ --
  [assert  base @ #10 =  assert]
;
```

Chapter 9

ANS Environment System

The file COMMON\ENVIRON.FTH provides the ANS ENVIRONMENT? word, and some basic environment data.

`Vocabulary Environment \ used for enviroment? queries`

The vocabulary within which environment data is kept.

`: ENVIRONMENT? \ c-addr u -- false | i*x true`

The string is treated as a word name. If it is found in the ENVIRONMENT vocabulary, the word is executed the results of executing it plus true are returned, otherwise just false is returned.

Chapter 10

Software Floating Point

10.1 Introduction

Although most embedded applications only require integer arithmetic, some do require floating-point. Therefore software floating-point is supplied with the cross-compiler and the target Forth. The target floating point wordset is not fully ANS compliant, but satisfies the needs of embedded systems without undue complexity. The Forth data stack and the floating point stack are the same. The floating point data storage format is not IEEE format, but is optimised for performance on small controllers. If you need a separate floating point stack or IEEE format storage, please contact MPE. Any variations in the implementation will be documented in the target specific section of the manual.

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

10.2 Source code

The source code is in two sets of files, one for 32 bit Forth targets, the other for 16 bit targets. The files are:

```
COMMON\SFP32HI    32 bit primitives
COMMON\SFP32COM   32 bit high level code
COMMON\SFP16HI    16 bit primitives
COMMON\SFP16COM   16 bit high level code
```

These files use no assembler definitions. Some targets have code versions of the primitives, and these will be found in the CPU specific code directory. A significant increase in performance can be obtained by using the code files.

10.3 Entering floating-point numbers

Floating-point numbers can be entered in two forms, 1.234 and 0.1234e1. Floating-point numbers are compiled as literal numbers when in a colon definition and placed on the cross-compiler's stack when outside a definition.

10.4 The form of floating-point numbers

A floating-point number is placed on the Forth data stack. In the Forth literature, this is referred to as a combined floating point and data stack. For 32 bit targets, a floating point number consists of two 32-bit numbers, one for the mantissa and one for the exponent. For 16 bit targets, it consists of a 32-bit double mantissa and a single 16-bit exponent. The mantissa is normalised. The exponent is on the top of the stack. Note that for 16 bit targets, number conversion is affected by the cross-compiler directives HOST-MATH and TARGET-MATH. HOST-MATH leaves double numbers and floats in 32-bit form, whereas TARGET-MATH leaves them in 16-bit form.

10.5 Creating variables

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

```
FVARIABLE VAR1
```

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

10.6 Accessing variables

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

10.7 Creating constants

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

```
1.234 FCONSTANT CON1
```

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

10.8 Using the supplied words

The supplied words split into several groups:

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

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

10.8.1 Calculating sines, cosines and tangents

To calculate sine, cosine and tangent, use FSIN, FCOS and FTAN respectively. Angles are expressed in radians.

10.8.2 Calculating arc sines, cosines and tangents

To calculate arc sine, cosine and tangent, use FASIN, FACOS

and FATAN respectively. They return an angle in radians.

10.8.3 Calculating logarithms

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

10.8.4 Calculating powers

Three power functions are supplied:

```
FE^X F10^X X^Y
```

10.9 Degrees or radians

The angular measurement used in the trigonometric functions are in radians. To convert between degrees and radians use RAD>DEG or DEG>RAD. RAD>DEG converts an angle from radians to degrees. DEG>RAD converts an angle from degrees to radians.

10.10 Displaying floating-point numbers

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

10.11 Changes from v6.0 to v6.1

Renamed DINT to F>D for consistency. F>D is the ANS word. The original F>D was just a synonym. Similarly SINT was renamed to F>S.

The word FLOATS that enabled floating point number conversion has been renamed to REALS to avoid a name conflict with the ANS word of the same name.

The F-PACK vocabulary has been removed as no one liked it, and it could be considered contrary to the ANS Forth specification. If you wish to retain the F-PACK vocabulary, add the following lines before and after the compilation of the floating point code:

```
only forth definitions          \ *** added ***
vocabulary f-pack               \ *** added ***
also f-pack definition          \ *** added ***
include %CommonDir%\Sfp32Hi     \ primitives
include %CommonDir%\Sfp32Com    \ common high level code
previous definitions            \ *** added ***
```

The code enabling floating point to work in degrees or radians has been commented out for ANS compatibility. All trig functions now operate in radians. The commented out code may be uncommented if you need backward compatibility.

10.11.1 32 bit targets: software floating point

Overhauled 32 bit software floating point and incorporated improvements contributed by Hiden Analytical. These include more complete special case detection, faster high level code, and more accurate number input and output.

Removed all use of global variables except PLACES to make the floating point code usable in interrupt routines and in multitasked systems. If the output routines are to be multitasked, change the definition of PLACES from:

```
VARIABLE PLACES 8 PLACES !
```

to:

```
CELL +USER PLACES
```

and remember to initialise PLACES before using the floating point output routines.

Many words that are only useful as factors have been made headerless to save target memory space.

10.11.2 16 bit targets: software floating point

Note that the 16 bit floating point pack is not re-entrant. If you need to use the floating point pack in a multitasking system, you should convert the global variables to USER variables. The word +USER can be used

```
<size> +USER <name>
```

to define a USER variable of a given size (normally a CELL) at the next free offset in the USER area. Only PLACES will need initialisation.

10.12 Glossary

10.12.1 Basic stack and memory operators

: `F! \ r addr --`

Stores r at addr

: `F@ \ addr -- r`

Fetches r from addr.

: `F, \ r --`

Lays a real number into the dictionary, reserving 8 bytes.

: `FDUP \ r -- r r`

Floating point equivalent of DUP.

: `FOVER \ r1 r2 -- r1 r2 r1`

Floating point equivalent of OVER.

: `FROT \ r1 r2 r3 -- r2 r3 r1`

Floating point equivalent of ROT.

: `FPICK \ fu..f0 u -- fu..f0 fu`
Floating point equivalent of PICK.

: `FROLL \ f1 f2 f3 -- f2 f3 f1`
Floating point equivalent of ROLL.

: `FSWAP \ r1 r2 -- r2 r1`
Floating point equivalent of SWAP.

: `FDROP \ r --`
Floating point equivalent of DROP.

: `FNIP \ r1 r2 -- r2`
Floating point equivalent of NIP.

10.12.2 Floating point defining words

: `FVARIABLE \ "<spaces>name" -- ; Run: -- f-addr`
Use in the form: FVARIABLE <name> to create a variable that will hold a floating point number.

: `FCONSTANT \ r "<spaces>name" -- ; Run: -- r`
Use in the form: <float> FCONSTANT <name> to create a constant that will return a floating point number.

: `FARRAY \ "<spaces>name" fn-1..f0 n -- ; Run: n -- rn`
Use in the form: n FARRAY <name> to create a variable that will hold a default floating point number. When the array name is executed, the index i is used to retun the address of the i'th 0 zero-based element in the array. For example, 5 FARRAY TEST will set up 5 array elements each containing 0, and then f n TEST F! will store f in the nth element, and n TEST F@ will fetch it.

10.12.3 Type conversions

: `NORM \ n exp -- f`
Normalise a single integer and a single exponent to produce a floating point number. INTERNAL.

: `DNORM \ d exp -- fn ; normalise a 64 bit double`
Normalise a double integer and a single exponent to produce a floating point number. INTERNAL.

: `FSIGN \ fn -- |fn| flag ; true if negative`
Return the absolute value of fn and a flag which is true if fn is negative.

: `F>S \ fn -- n`
Converts a float to a single integer. Note that F>S truncates the number towards zero according to the ANS specification. If —fn— is greater than maxint, +/-maxint is returned.

: `F>D \ fn -- d`
Converts a float to a single integer. Note that F>D truncates the number towards zero according to the ANS specification. If —fn— is greater than dmaxint, +/-dmaxint is returned.

: `FINT \ f1 -- f2`
Chop the number towards zero to produce a floating point representation of an integer.

: `S>F \ n -- fn`
Converts a single integer to a float.

: `D>F \ d -- fn`
Converts a double integer to a float.

10.12.4 Arithmetic

: `FNEGATE \ r1 -- r2`
Floating point negate.

: `?FNEGATE \ fn n -- fn|-fn`
If n is negative, negate fn.

: `FABS \ fn -- |fn|`
Floating point absolute.

: `F* \ r1 r2 -- r3`
Floating point multiply.

: `F/ \ r1 r2 -- r3`
Floating point divide.

: `F+ \ r1 r2 -- r3`
Floating point addition.

: `F- \ r1 r2 -- r3`
Floating point subtraction.

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

: `FFRAC \ f1 f2 -- f3`
Leave the fractional remainder from the division f1/f2. The remainder takes the sign of the dividend.

10.12.5 Relational operators

`: F0< \ f1 -- flag`

Floating point 0<.

`: F0> \ f1 -- flag`

Floating point 0>.

`: F0= \ f1 -- flag`

Floating point 0=.

`: F0<> \ f1 -- flag`

Floating point 0<>.

`: F= \ f1 f2 -- flag`

Floating point =.

`: F< \ r1 r2 -- flag`

Floating point <.

`: F> \ f1 f2 -- flag`

Floating point >.

`: FMAX \ r1 r2 -- r1|r2`

Floating point MAX.

`: FMIN \ r1 r2 -- r1|r2`

Floating point MIN.

10.12.6 Rounding

`f# 1.0 fconstant %ONE`

Floating point 1.0.

`: FLOOR \ r1 -- r2`

Floored round towards -infinity.

`: FROUND \ r1 -- r2`

Round the number to nearest or even.

10.12.7 Miscellaneous

`: FALIGNED \ addr -- f-addr`
Aligns the address to accept an 8-byte float.

`: FALIGN \ --`
Aligns the dictionary to accept an 8-byte float.

`: FDEPTH \ -- +n`
Returns the number of floats on the stack.

`: FLOAT+ \ f-addr1 -- f-addr2`
Increments addr by 8, the size of a float.

`: FLOATS \ n1 -- n2`
Returns n2, the size of n1 floats.

10.12.8 Floating point output

`1 s>f 10 s>f f/ fconstant %.1`
Floating point 0.1.

`1 s>f fconstant %1`
Floating point 1.0.

`10 s>f fconstant %10`
Floating point 10.0.

`1250000000 34 fconstant %10^10`
Floating point 10^10.

`1844674407 -33 fconstant %10^-10`
Floating point 10^-10.

`F# 1.0E256 FCONSTANT %10^256`
Floating point 10^256.

`F# 1.0E-1 FCONSTANT %10E-1`
Floating point 10^-1.

`F# 1.0E-10 FCONSTANT %10E-10`
Floating point 10^-10.

`F# 1.0E-256 FCONSTANT %10^-256`
Floating point 10^-256.

`16 FARRAY POWERS-OF-10E1`

An array of 16 powers of ten starting at 10^0 in steps of 1.

`17 FARRAY POWERS-OF-10E16`

An array of 17 powers of ten starting at 10^0 in steps of 16.

`16 FARRAY POWERS-OF-10E-1`

An array of 16 powers of ten starting at 10^0 in steps of -1.

`17 FARRAY POWERS-OF-10E-16`

An array of 17 powers of ten starting at 10^0 in steps of -16.

`: RAISE_POWER \ mant exp -- mant' exp'`

Raise the power in preparation for number formatting.

`: SINK_FRACTION \ mant exp -- mant' exp'`

Reduce the power in preparation for number formatting.

`variable places 8 places ! \ -- addr`

Number of digits output after the decimal point.

`: ROUND \ f1 -- f2`

Rounds least significant eight bits to 0 if higher 2 bits are all 0s or all 1s.

`: ?10PWR \ exp[2] -- exp[2] exp[10]`

Generate the power of ten corresponding to the power of two. INTERNAL.

`: SIGFIGS \ fn n -- d dec_exponent`

From fn, generate a double number corresponding to n significant digits and a decimal exponent. INTERNAL.

`: op-prepare \ fn -- d exp sign`

From fn, generate a double number corresponding to n significant digits, a decimal exponent and a sign indicator (nz=negative). INTERNAL.

`: .EXP \ exp --`

Display the exponent. INTERNAL.

`: N# \ d n -- d'`

Convert n digits. INTERNAL.

`: E. \ n exp --`

Print the f.p. number on the stack in exponential form, x.xxxxxEyy.

: REPRESENT \ r c-addr u -- n flag1 flag2

Assume that the floating number is of the form +/-0.xxxxEyy. Place the significand xxxxx at c-addr with a maximum of u digits. Return n the signed integer version of yy. Return flag1 true if f is negative, and return flag2 true if the results are valid. In this implementation all errors are handled by exceptions, and so flag2 is always true.

: F. \ f --

Print the f.p. number in free format, xxxx.yyyy, if possible. Otherwise display using the x.xxxxEyy format.

10.12.9 Floating point input

: FLITERAL \ Comp: r -- ; Run: -- r

Compiles a float as a literal into the current definition. At execution time, a float is returned. For example, [%PI F2*] FLITERAL will compile 2PI as a floating point literal. Note that FLITERAL is immediate.

: CONVERT-EXP \ c-addr --

If the character at c-addr is 'D' convert it to 'E'. INTERNAL.

: CONVERT-FPCHAR \ c-addr --

Convert the f.p. char '.' to the double char ',' for conversion. INTERNAL.

: ALL-BLANKS? \ c-addr len -- flag

Return true if string is all blanks (spaces). INTERNAL.

: FCHECK \ -- am lm ae le e-flag .-flag

Check the input string at PAD, returning the separated mantissa and exponent flags. The e-flag is returned true if the string contained an exponent indicator 'E' and the .-flag is returned true if a '.' was found. INTERNAL.

: MNUM \ c-addr u -- d 2 | 0

Convert the mantissa string to a double number and 2. If conversion fails, just return 0. INTERNAL.

: ENUM \ c-addr u -- n 1 | 0 ; str as above

Convert the mantissa string to a single number and 1. If conversion fails, just return 0. INTERNAL.

: *10^X \ float dec_exponent -- float'

Generate float' = float *10^dec_exp. INTERNAL.

: `FIXEXP \ dmant exp -- mant' exp'`

Convert a double integer mantissa and a single integer exponent into a floating point number. INTERNAL.

: `FNUMBER? \ addr -- 0/.../mant exp 2`

Behaves like the integer version of NUMBER? except that if the number is in F.P. format and BASE is decimal, a floating point conversion is attempted. If conversion is successful, the floating point number is left on the float stack and the result code is 2.

: `>FLOAT \ c-addr u -- r true|false`

Try to convert the string at c-addr/u to a floating point number. If conversion is successful, flag is returned true, and a floating number is returned on the float stack, otherwise just flag=0 is returned.

: `(F#) \ addr -- fn 2 | 0`

The primitive for F# and F#IN below.

: `F#IN \ -- fn 2 | 0`

Attempts to convert a token from the input stream to a floating-point number. Numbers in integer format will be converted to floating-point. An indicator (0 or 2/3) is returned in the same way as an indicator is returned by FNUMBER?.

: `F# \ -- [f] ; or compiles it [ state smart ]`

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

: `REALS \ -- ; allow f.p input`

Switch NUMBER? to permit floating point input using FNUMBER?. This action can be reversed by INTEGERS. Both REALS and INTEGERS are in the FORTH vocabulary.

: `INTEGERS \ -- ; no f.p input`

Switch NUMBER? to restore integer only input.

10.12.10 Trigonmetric functions

N.B. All angles are in radians.

: `DEG>RAD \ n1 -- n2`

Convert degrees to radians.

: `RAD>DEG \ n1 -- n2`

convert radians to degrees.

: `FSIN \ f1 -- f2`
f2=sin(f1).

: `FCOS \ f1 -- f2`
f2=cos(f1).

: `FTAN \ f1 -- f2`
f2=tan(f1).

: `FASIN \ f1 -- f2`
f2=arcsin(f1).

: `FACOS \ f1 -- f2`
f2=arccos(f1).

: `FATAN \ f1 -- f2`
f2=arctan(f1).

10.12.11 Power and logarithmic functions

: `FLN \ f1 -- f2`
Take the logarithm of f1 to base e and return the result.

: `FLOG \ f1 -- f2`
Take the logarithm of f1 to base 10 and return the result.

: `FE^X \ f1 -- f2`
f2=e^f1.

: `F10^X \ f1 -- f2`
f2=10^f1

: `FX^N \ x-real n-integer -- fx^n`
fx^n=x^n where x is a float and n is an integer.

: `FX^Y \ x-real y-real -- fn`
fn=X^Y where Y and Y are both floats.

: `FSQR \ f1 -- f2 ; FSQR by Heron's formula`
F2=sqrt(f1) by Heron's formula.

10.13 High Level primitives

The software floating point pack requires several support primitives. High level versions are provided in SFP16HI.FTH and SFP32HI.FTH for 16 and 32 bit targets. Some targets have coded versions in the CPU directory and these will provide much better performance. The support file should be compiled before the common file.

`: <<1 \ n -- n<<1`

A compiler synonym for 2* or "1 LSHIFT".

`: >>1 \ n -- n>>1`

A compiler synonym for 2/ or "1 RSHIFT".

`: S-> \ n1 carry-in-flag --- n2 carry-out-flag`

Perform a right shift, applying the carry in to the m.s. bit and returning the carry out as 1 or 0.

`: <-S \ n1 carry-in-flag --- n2 carry-out-flag`

Perform a left shift, applying the carry in to the l.s. bit and returning the carry out as 1 or 0.

`: d<<1 \ xd -- xd<<1`

One bit double left shift.

`: d>>1 \ xd -- xd>>1`

One bit double right shift.

`: D>>N \ d m -- d>>m`

M bit double right shift.

Chapter 11

Periodic Timers

This code provides a timer system that allows many timers to be defined, all slaved from a single periodic interrupt. The Forth words in the user accessible group documented below are compatible with the token definitions for the PRACTICAL virtual machine, with the code supplied with MPE's embedded targets, and with VFX Forth. This code assumes the presence of a global value TICKS which holds a time value incremented in milliseconds. The timebase is approximate, and granularity and jitter are affected by the timer ISR and the time taken by your own code to execute. By default, the timer is set to run every 100ms. The source code is in the the file TIMEBASE.FTH.

The timer chain is built using a buffer area, and two chain pointers. Each timer is linked into either the free timer chain, or into the active timer chain.

All time periods are in milliseconds. Note that on a 32 bit system such as ProForth VFX, these time periods must be less than 2^31-1 milliseconds, say 596 hours or 24 days, whereas if the code is on a 16 bit system, time periods must be less than 2^15-1 milliseconds, say 32 seconds.

11.1 The basics of timers

These basic words are defined for applications to use the timer system. Other words are detailed elswhere in this chapter.

```
START-TIMERS    \ -- ; must do this first
STOP-TIMERS     \ -- ; closes timers
AFTER           \ xt period -- timerid/0 ; runs xt once after period ms
EVERY           \ xt period -- timerid/0 ; runs xt every period ms
TSTOP           \ timerid -- ; stops the timer
MS                      \ period -- ; wait for period ms
```

After the timers have been started, actions can be added. The example below starts a timer which puts a character on the debug console every two seconds. Note that when using generic I/O, the output and input devices MUST be specified.

```
start-timers
: t     \ -- ; will run every 2 seconds
  console opvec !
  [char] * emit
;
' t 2 seconds every             \ returns a timer id, use TSTOP to stop it
```

The item on stack is a timer handle, use TSTOP to halt this timer.

AFTER is very useful for creating timeouts, such as required to determine if something has happened in time. AFTER returns a timerid. If the action you are protecting happens in time, just use TSTOP when the action happens, and the timer will never trigger. If the action does not happen, the timer event will be triggered.

11.2 Considerations when using timers

All timers are executed within a single interrupt, and so all timer action words share a common user area. This has some impact on timer action words. Since you do not know in which order timer action words are executed, you must set up any USER variables such as BASE that you may use, either directly or indirectly.

The interrupt that handles all the timers does not set IPVEC and OPVEC to a default value. If you are going to use Forth I/O words such as EMIT and TYPE within a timer action, you MUST set IPVEC and OPVEC before using the I/O. For the sake of other timer action routines that may still be using default I/O, it is polite to save and restore IPVEC and OPVEC in your timer action words.

Do not worry about calling TSTOP with a timerid that has already been executed and removed from the active timer chain; if TSTOP cannot find the timer, it will ignore the request.

Under some conditions, the execution time of all the timer routines may be longer than the requested period of the timer. In addition, the timer interrupt may be subject to jitter.

11.3 Implementation issues

The following discussion is relevant if you want to modify this code. Functionally equivalent code is provided with MPE's VFX Forth systems. In the Windows environment, timer interrupts are implemented by callbacks and critical sections.

By default, the word DO-TIMERS is run from within the periodic timer interrupt. If interrupts are not re-enabled after resetting the timer interrupt, you may have latency issues if a number of timers is used, or if the timer routines take a considerable time. In this case, it would be better to set up the timer routine to RESTART a task, which calls DO-TIMERS, e.g.

```
: TIMER-TASK    \ --
  <initialise>
  BEGIN
    DO-TIMERS STOP
  AGAIN
;
```

Such a strategy also permits you to use a fast interrupt, say 1ms, for the clock, and to trigger the TIMER-TASK every say 32 ms.

11.4 Timebase glossary

`0 value ticks \ -- addr ; holds timer count`
Get current clock value in milliseconds.

`#8 constant #timers \ -- n ; maximum number of timers`
A constant used at compile time to set the maximum number of timers required. Each timer requires RAM as defined by the ITIMER structure.

`: after \ xt period -- timerid/0 ; xt is executed once,`
Starts a timer that executes once after the given period. A timer ID is returned if the timer could be started, otherwise 0 is returned.

`: every \ xt period -- timerid/0 ; xt is executed periodically`
Starts a timer that executes every given period. A timer ID is returned if the timer could be started, otherwise 0 is returned. The returned timerID can be used by TSTOP to stop the timer.

`: tstop \ timerid --`
Removes the given timer from the active list.

`: pause \ -- ; multitasker hook`
Allows the sytem multitasker to get a look in. Under Windows this also allows the message queue to be handled.

`: later \ n -- n'`
Generates the timebase value for termination in n millseconds time.

`: expired \ n -- flag ; true if timed out`
Flag is returned true if the timebase value n has timed out. N.B. Calls PAUSE.

`: timedout? \ n -- flag ; true if timed out`
Flag is returned true if the timebase value n has timed out. Does not call PAUSE, so can be used in interrupts, winprocs and callbacks. In particular, TIMEDOUT? should be used rather than EXPIRED inside timer action words to reduce timer jitter.

`: ms \ n --`
Waits for n milliseconds. Uses PAUSE through EXPIRED.

Chapter 12

Vocabulary and wordlist tools

: `VOC? \ wid -- flag`

Return TRUE if 'wid' is actually a vocabulary.

: `.VOC \ wid --`

If wid represents a vocabulary, display its name, otherwise just display its value.

: `ORDER \ --`

Display the current search order and definitions vocabularies.

: `VOCS \ --`

Display all vocabularies.

: `$FORGET \ c-addr --`

Forgets word name in given string. See FORGET.

: `FORGET \ "<spaces>name" --`

Used in the form "FORGET <name>", <name> and all following words are removed from the dictionary. This word is marked obsolescent in the ANS specification, and is replaced by MARKER.

: `MARKER \ "<spaces>name" -- ; Exec:  --`

MARKER <name> creates a word that when executed removes itself and ALL following definitions from the dictionary. MARKER is the ANS replacement for FORGET. MARKER automatically trims all wordlist and vocabulary based chains.

Chapter 13

XMODEM Receiver and Transmitter

13.1 Introduction

This file implements the XMODEM 128-byte protocol in both directions. Use with AIDE requires AIDE release 2.500 upwards.

No test code is provided for this file as the system has been tested by comparison of transferred binary files.

13.2 Words in XMODEMTXRX.FTH

13.2.1 Configuration

`1 equu XmodemTx?  \ -- n`

Non-zero to compile transmit code

`1 equ XmodemRx?  \ -- n`

Non-zero to compile receive code.

13.2.2 Constants and variables

`$0101 equ blkerror`

A block number error has occurred.

`$0103 equ noreply`

There was no reply within one second.

`$0104 equ crcerror`

Bad CRC or checksum.

```
$0105 equ overflow
```

Too many blocks were sent.

```
$010 equ maxerrs \ -- n
```

Maximum number of errors before transfer is aborted.

```
#128 Buffer:  x-buffer \ -- addr
```

Holds a 128 byte Xmodem data block.

13.2.3 Common code

```
:  init-blks \ addr #bytes -- #blks
```

Given the size of an image, return the number of complete 128 byte blocks, set the variable CUR-ADDRESS to ADDR and set the variable BLK# to 1.

13.2.4 XMODEM transmission

```
:  (To-Buffer) \ -- ; copy 128 bytes to X-BUFFER
```

Move 128 bytes from the memory pointed to by CUR-ADDRESS into X-BUFFER. This is the default action of TO-BUFFER. The variable CUR-ADDRESS is set by BIN-DOWN and friends.

```
Defer To-Buffer \ --
```

Copy the next 128 bytes to transmit to X-BUFFER. They are then transmitted from X-BUFFER. You can change this action as required by your application. The default action is (TO-BUFFER).

```
:  ?Ack \ -- ; wait for char, abort if not ACK
```

Wait for a character and terminate the transfer and abort if the character is not an ACK.

```
:  Send-Block \ Blk# -- ; transmit a block
```

Transmit the 128 byte contents of X-BUFFER to the host.

```
:  Bin-Down \ addr #bytes -- ; transfer memory to host
```

Download (transmit) the given block of memory to the host using the XMODEM 128 byte block protocol. On entry, the variable CUR-ADDRESS is set to ADDR on entry and #bytes is rounded up to a 128 byte unit.

13.2.5 XMODEM reception

`: ser-flush \ -- ; flush the link input`
Flush all input character from the host/target link.

`: send-ack \ -- ; send ACK`
Transmit an ACK character.

`: send-nak \ -- ; Transmit a NAK character`
Transmit a NAK character.

`: send-can \ -- ; send CAN character`
Transmit a CAN character.

`: Get-BlockData \ -- cksum ; get block from host`
Receive a 128 byte XMODEM data block from the host.

`: toomanyerrs? \ -- T|F ; true if too many errors`
Return true if too many comms errors have occurred.

`: (From-Buffer) \ -- ; copy 128 bytes from X-BUFFER`
Move 128 bytes from X-BUFFER into the memory pointed to by CUR-ADDRESS

`Defer From-Buffer \ -- ; copy 128 bytes from X-BUFFER`
Move 128 bytes from X-BUFFER into the memory pointed to by CUR-ADDRESS. CUR-ADDRESS is set up by BIN-UP and friends. The default action is (FROM-BUFFER). You can modify the action to suit your own application.

`: Get-Block \ -- ;`
Receive an XMODEM 128 byte data block from the host, processing the header and checksum data.

`: waitforresponse \ -- ; wait for host to respond`
Wait for the host to respond for up to 1 second. If the host does not respond, a NOREPLY status is set in the variable RXSTATUS. This word requires the ticker interrupt to be running.

`: Bin-Up \ addr len -- status ; status 0 = GOOD`
Upload (receive) a block of data of the given size into memory using the XMODEM 128 byte block protocol. An error status is returned, 0 indicating success. On entry, the variable CUR-ADDRESS is set to ADDR on entry and len is rounded up to a 128 byte unit. Note that an error return of $0105 indicates the the file being sent is larger than the LEN input parameter.

`: RecvXmodem \ addr len -- len' status ; status=0=good`
Upload (receive) a block of data of the given size into memory using the XMODEM 128 byte block protocol. The number of bytes correctly received and an error status are returned, 0 indicating success. See BIN-UP above for more details.

Chapter 14

ROM PowerForth utilities

14.1 Introduction

Supplied as source in the ROMFORTH directory are utilities to:

- compile source code on your target board from the cross-compiler IDE
- upload a binary image from your target to your PC
- download a binary image to your target from your PC.

Note that the target source code supplied with cross compiler versions 6.02 onwards is incompatible with code supplied for previous versions of the cross compiler.

These utilities can be used to generate an EPROM that has all the tools required to develop an application, or can be used during development to transfer modules to and from your PC. All the code is designed to be used with the MPE development environment, AIDE. The code will also work with other compatible terminal emulators.

Users who wish to distribute ROMs containing the ROM PowerForth utilities should contact MPE for details of the OEM licence, which includes documentation on disc of the Forth kernel and the ROM PowerForth utilities.

14.2 Compiling text files

Source text files can be compiled from the host PC onto the target system. This saves time in not having to cross-compile the entire source if a small modification is made. The utilities permit text file to be split into pages for better layout when printed. An ASCII Form Feed character (decimal 12) separates one page from another.

14.2.1 The required files

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

14.2.2 Compiling a specified text file

To compile all or part of a specified text file onto your target, use GET or INCLUDE in the form:

```
INCLUDE <filename>
```

This compiles the file <filename> into the target's dictionary. AIDE's internal file server must be enabled (in the console window configuration), and will be triggered automatically.

14.3 Downloading a binary image

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

- Intel hex download
- XMODEM download

For both utilities the cross-compiler IDE or a suitable communications package will be required.

14.3.1 XMODEM binary image download

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

Required files To use this utility you must cross-compile the file COMMON\XMODEMTXRX.FTH.

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

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

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

```
1200 400 BIN-DOWN
```

sends the area of memory from 1200 to 1599 to your host PC. AIDE's internal file server must be enabled (in the console window configuration), and will be triggered.

14.3.2 Intel hex download

Binary images can be downloaded to your PC using the Intel hex format. Because this format only uses the standard printable character and CR/LF, the data can be captured by very simple tools.

Required files To use this utility you must cross-compile the file INTELHEX.FTH.

Using the Hex download utility To download a binary image from the target system to your PC, use HEX-DOWN in the form:

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

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

```
1200 400 HEX-DOWN
```

sends the area of memory from 1200 to 1599 to your host PC. In AIDE, turn on console logging to receive the file. In other packages this may be referred to as file capture.

14.4 ROM PowerForth

ROM PowerForth can be used to generate a stand-alone Forth system. With these utilities, you can generate an EPROM that contains an interactive Forth with the ability to develop an application. Note: A licence is required to distribute open Forth systems. Contact MPE for more details.

14.4.1 Hardware requirements

To develop an application using ROM PowerForth, your board requires an area which:

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

14.4.2 EPROM/Flash area

The area that is always EPROM contains the development kernel.

14.4.3 RAM area

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

14.4.4 RAM/EPROM area

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

14.4.5 Types of board

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

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

Three site boards The three areas are provided by three memory sockets:

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

Two site boards with battery backed RAM The three areas are provided by two sockets:

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

Two site boards with socket converter On many boards, there is unused space in the EPROM as ROM PowerForth occupies less than 32k bytes of memory. Therefore, a header board can be made which converts one socket into two. For example, if the socket normally takes a 27512 EPROM, a board can be made which has a 32k EPROM with the ROM PowerForth development kernel and 32k bytes of RAM. To access the RAM, the write line is attached to a suitable point on the main board with a fly lead. After the application has been developed, the two images are combined back into a single EPROM.

14.4.6 Making your application turnkey

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

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

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

```
' JOB SETUP
```

14.4.7 Discarding the application RAM area

The application can be discarded by typing:

```
0 ROM !
```

14.4.8 Changing the application RAM start address

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

14.4.9 AIDE file server protocols

AIDE's file server must be enabled for automatic file handling. Details of the protocols used should be obtained from this manual and the source code in the COMMON\ROMFORTH directory.

14.5 IODEF.FTH

Before compiling this file, synonyms may need to be defined for SER-EMIT SER-KEY? and SER-KEY. Add these in the control file before compiling IODEF.FTH.

IODEF.FTH provides equates and protocol primitives for AIDE.

14.5.1 AIDE support

`variable disk-error \ -- addr ; set non-zero on error`

This variable is set true when a transfer error occurs.

`: wait-ack \ -- ; wait for ACK character`

This word waits for the host to send an ACK at the end of part of a transfer. INTERNAL.

`: wait-ack/nack \ -- t/f ; true for NACK`

This waits for either a NACK or an ACK from the host and leaves true or false on stack. INTERNAL.

`: send-block# \ n -- ; send block number to server`

Sends a single length number as two bytes 00-FF, low byte first. INTERNAL.

`: synch-to-host \ -- ; sync host to us`

Waits for a START (0x01) character, flushes the input, and sends an ACK. INTERNAL. INTERNAL.

14.6 Miscellaneous

`: cls \ --`

Clear PowerTerm screen

14.7 Application Extensions

ROM PowerForth can be used to generate a stand-alone Forth system. With these utilities, you can generate an EPROM that contains an interactive Forth with the ability to develop an application. Note: A licence is required to distribute open Forth systems. Contact MPE for more details.

The code in COMMON\ROMFORTH\LINK.FTH provides the facilities to link the extension area into the ROM PowerForth kernel.

`appl-rom equ ROM \ start of ROM/RAM area`

Define the address of a ROM/RAM area in which applications are to be developed. This area is RAM during development. When development is complete, the word SETUP is executed to initialise the memory area. The compiled image is then downloaded to the PC using the tools in the ROMFORTH directory. The image is then programmed into EPROM, which can then replace the RAM. If the word RELINK is added to COLD, the application EPROM will be automatically linked in when the target powers up.

`variable rp EM rp ! \ top of RAM area`

If your processor has separate CODE and DATA address spaces, it has a HARVARD architecture. This word is only compiled for Harvard targets such as 8051s. The working RAM area has the interactive RAM dictionary at the bottom and the application buffer space at the top. Define the initial value of RP to be the highest available free location in the RAM area. Application variable and buffer space will then be allocated downwards from this address. When creating an application to be ROMmed, download the code for BUFFER and the redefined VARIABLE first of all.

`: setup \ xt|0 -- ; sets up for ROM generation`

Use XT as the xt of the word to be run when the extension is found.

`: use-setup-data \ --`

Apply the data in the setup area.

`: link-rom \ --`

Link in the application ROM area.

`: link-ram \ -- ; link in application RAM area`

Link in the application RAM area.

`: link-nv-ram \ -- ; link in for NV-RAMs`

Link in the application NV-RAM area.

`: relink \ -- ; re-link application if there`

Link in the application ROM/RAM/NV-RAM area.

14.8 INCLUDE source code from AIDE

The file COMMON\ROMFORTH\TEXTFILE.FTH provides support for compiling a source file from the AIDE server. The code has been updated for AIDE version 2.500 onwards.

`: end-load \ -- ; switch back to keyboard input`

This word is automatically performed at the end of a download to tidy up the comms.

`: file-error \ n --`

Handle an error when a file is being INCLUDEd.

`: $include \ $addr -- ; compile host file, counted string`

Given a counted string representing a file name, compile the file from AIDE.

`: include \ "<filename>" -- ; load file from host`

Compile a file across the serial line from the AIDE file server. Use in the form:

```
include <filename>
```

The filename extension must be supplied.

14.9 Simple source file loader

The code in COMMON\ROMFORTH\FILETRAN.FTH provides a simple source file loader which can be used with most terminal emulators. The download is controlled by XON/XOFF flow control. When using the PowerTerm terminal emulator in AIDE, use the INCLUDE <filename> system which supports nested files and needs no special termination.

Each file compiled must include a single line

```
END-UP-LOAD
```

at the end to reset the interpreter.

For slow 8 bit CPUs without queued serial input, the terminal server may need to include pacing delays after each character and an additional after CR/LF pairs.

`: Up-Load \ -- ; Load ASCII text`

Compile a file delivered by the terminal emulator. This word is intolerant of compilation errors.

`: End-Up-Load \ -- ; Finish Up-Loading`

Used on the target to restore the Forth interpreter after a file has been compiled.

14.10 Intel Hex transfers

The code in COMMON\ROMFORTH\INTELHEX.FTH provides a simple way to transfer binary images from ROM PowerForth to a host. The Intel Hex format is printable and can be captured (logged) by most terminal emulators including AIDE.

`: HEX-DOWN \ addr #bytes --`

Send #bytes at addr from ROM PowerForth to the host in Intel Hex format.

14.11 Block support

BLOCKS.FTH provides support for the transfer of 1k blocks between a host PC and the embedded target. Although the use of blocks as source screens is now obsolete, blocks are still useful for the transfer of binary data, especially for data loggers which use paged RAM or Flash for data storage. AIDE version 3 or above is required.

Despite the comment about the obsolescence of screens, the code in BLOCKS.FTH provides screen file support for legacy systems.

14.11.1 Primitives

`: BLK-READ \ addr blk# --`

Reads one block from host to addr.

`: BLK-WRITE \ addr blk# --`

Sends given block number to host.

14.11.2 Application words

`$400 equ /block \ -- size ; size of a block`

The size of a block.

`/first buffer: FIRST \ -- addr ; first element is block no.`

The data buffer. The first cell contains the block number and is followed by /BLOCK data bytes.

`: SAVE-BUFFERS \ --`

Save the data buffer if it has been modified.

`: EMPTY-BUFFERS \ --`

Mark the data buffer as empty.

`: UPDATE \ --`

Mark the data buffer as modified.

`: FLUSH \ --`

Force a changed buffer to be uploaded to the host.

`: BLOCK \ u -- addr`

Addr is the address of the first character of the block buffer assigned to mass-storage block u. An ambiguous condition exists if u is not an available block number. If block u is already in a block buffer, a-addr is the address of that block buffer. If block u is not already in memory and there is an unassigned block buffer, transfer block u from mass storage to an unassigned block buffer. Addr is the address of that block buffer. If block u is not already in memory and there are no unassigned block buffers, unassign a block buffer. If the block in that buffer has been UPDATEd, transfer the block to mass storage and transfer block u from mass storage into that buffer. Addr is the address of that block buffer. At the conclusion of the operation, the block buffer pointed to by addr is the current block buffer and is assigned to u.

14.11.3 Block file management

: `-trailing \ addr len -- addr len' ; strip trailing spaces`
This word strips the trailing spaces from a string. It is most useful when displaying lines from a screen to strip the trailing spaces from the 64 byte fixed length lines.

: `.LINE \ line# blk# --`
Display line# from blk#.

: `?LOADING \ --`
THROW #-501 if input from console.

: `LIST \ blk# -- ; display screen given`
Display screen blk# as 16 lines of 64 characters.

: `L \ --`
LIST the current screen in SCR.

: `N \ --`
LIST the Next screen.

: `P \ --`
LIST the Previous screen.

: `Index \ n1 n2 --`
Display the top lines in the (inclusive) range of screens.

: `Qx \ --`
INDEX all available screens.

: `LOAD \ blk# -- ; compile given screen`
Save the current input-source specification. Store u in BLK (thus making block u the input source and setting the input buffer to encompass its contents), set >IN to zero, and interpret. When the parse area is exhausted, restore the prior input source specification. Other stack effects are due to the words LOADed.

: `THRU \ first last --`
LOAD screens from first to last in that order.

: `--> \ --`
LOAD the next screen.

: `$using \ $addr --`
Select the file specified by the counted string at $addr as the current block/screen file.

: `using \ "<file>" --`
Use in the form "USING <filename>" to select the current block/screen file. The AIDE file server is required. AIDE takes care of creating non-existent files.

14.12 Target BUFFER: and VARIABLE

The definitions below reorganise the use of RAM for ROM PowerForth applications which are themselves to be put in Flash or EPROM.

The variable RP points to the top of available RAM, and is decremented by the amount of RAM required. The word BUFFER: allocates memory from this space, returning its base (low) address. Words such as VARIABLE can then be defined in terms of BUFFER.

`: Buffer: \ n -- ; -- addr`

Return the address of an n byte buffer. Use in the form:

`<size> BUFFER: <name>`

`: variable \ -- ; -- addr ; replacement`

Create a new variable. Use in the form:

`VARIABLE <name>`

14.13 Some simple tools

The file COMMON\ROMFORTH\TEXTFILE.FTH provides some simple application tools.

The words (#IN) and #IN are provided to collect numbers from the keyboard in decimal. (#IN) is useful because it returns the number of digits converted so that 0 digits indicates that the user just pressed <Enter>, rather than entering a valid number of value 0.

`: (#in) \ -- n #chars`

Read a decimal number from the keyboard returning the number and the number of digits converted.

`: #in \ -- n`

Collect a decimal number from the keyboard.

Chapter 15

Examples directory

The EXAMPLES directory contains much useful code, ranging from simple tools to fully documented extensions. The best way to use the EXAMPLES directory is to browse through the source code. If you want to modify the code, we recommend that you move it to become part of your own application directory structure.

15.1 Main directory

The following is a list of files as of November 2002.

CALENDAR.FTH A perpetual calendar by Christophe Lavarenne. A choice of calendars is provided.

COSINE.FTH Integer 14 bit cosine generation, suitable for 16 bit systems. Tested on an RTX2000.

DALLAS.Z80 Driver for Dallas smart watch. Derived from source code provided by Gerry Coe of Devantech Electronics (good low cost boards) and modified by MPE.

DEFINE.FTH Provides an example of using defining words in both the cross compiler and the target.

DOUBLES.HI This file implements double and some quad precision number support using the primitives of PowerForth and high level definitions. To obtain better performance some definitions should be coded. These are indicated in the source code.

HEXPAD.FTH Keypad read routine for hex matrix keypad. The example was written for an 8051 port using four input bits and four output bits.

MATH.FTH Miscellaneous math functions.

PRIMES.FTH Eratosthenes sieve - simple prime benchmark.

SINCOS.FTH Integer trig words from Kurt Heinz at Synics. These words provide a simple implementation of sine, cosine, and tangent functions.

TESTCODE.FTH A test harness for verifying the stack effect of of Forth words and phrases.

15.2 Contributions subdirectory

This directory contains code contributed by users for others to use, and MPE thanks the contributors.

The contents of this directory are untouched by MPE who provide no warranty at all on this code. Sorry about that.

AD.FTH 68HC11 A/D handler.

CW.FTH This program will display text in CW (Morse Code) upon either the system's console or the system's LEDs.

DATES.FTH Conversions between calendar date and Julian day number from ACM# 199. Forth Scientific Library Algorithm #22

HIDEN.FTH This code replaces REQUEST and SIGNAL in the MPE multitasker because they allow a task to lock a semaphore multiple times.

IEEE.FTH Converts between MPE software floating point format for 32 bit systems and IEEE 32 bit format.

LANDER.FTH Lunar Landing Simulation.

15.3 Drivers subdirectory.

29F0X0.FTH 29F010/40 Driver code assuming a 16 bit bus using 2 devices.

CANREAL.FTH This file provides a set of words to act has a hardware abstraction layer for the i82527 drivers when using the physical device on the MPE H8 Board.

I82527.FTH i82527 CAN Controller Device Driver.

DARTCTC.FTH Serial i/o drivers for Z80/64180 + DART + CTC.

KEYBRD.DRV Code for 4x4 matrix keyboard connected via the MPE User Interface Card containing an 8255 PIA.

LCD.DRV Code for Hitachi LMG6400PLGR LCD Display. This will drive the Hitachi display connected via the MPE User Interface Card containing an 8255 PIA at base address defined in USERBRD.DRV.

SCSI5380.FTH SCSI interface words for RTX-2000 with a 5380 SCSI controller.

SER2681.FTH 2681 serial driver. This driver was written for a Cavendish Automation board

SMC91C9X.FTH SMC9192/94/96 Ethernet Driver Code.

USERBRD.DRV Code for MPE User Interface Board Setup for card containing an 8255 PIA at base address 0F000h. A glossary can be found in USERBRD.TXT

15.4 I2C subdirectory

I2CLOAD.BLD Build file for other I2C files.

BCD.FTH BCD to binary conversion and back

I2CBASE.FTH I2C primitives. This file requires an I2C bit-banging I/O driver to have been compiled.

I2CNOTES.DOC I2C documentation in Word format.

DEVICES\8574DRV.FTH Driver for an 8574.

DEVICES\8583DRV.FTH Driver for an 8583.

DRIVERS\I2CVFXDRV.FTH Bit banging parallel port driver for VFX Forth for Windows.

15.5 SPI subdirectory

SPINOTES.DOC SPI documentation in Word format.

SPILOAD.BLD Build file that pulls in other SPI files.

PPDRV.FTH PC printer port access for VFX Forth for Windows.

SPIVFXDRV.FTH SPI primitives for VFX Forth for Windows. Requires PPDRV.FTH.

SPIBASE.FTH SPI byte read and write primitives. A lower level driver is required.

25LCDRV.FTH Driver for a Microchip 25LC series SPI EEPROM.