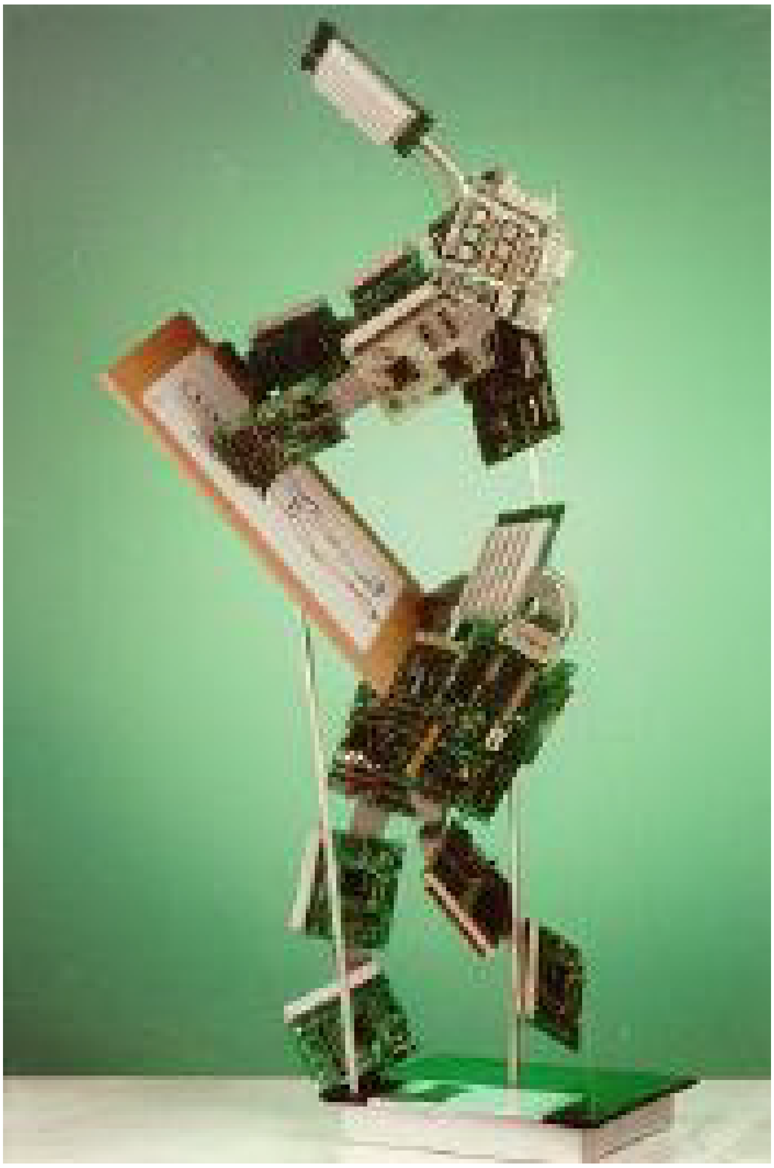
VFX Forth for x86/x86_64 Linux

Native Code ANS Forth Implementation



Microprocessor Engineering Limited

VFX Forth for Linux

Native Code ANS Forth Implementation

MPE VFX Forth for Linux

Published by Microprocessor Engineering

User manual
Manual revision 4.70
5 November 2014

Software
Software version 4.70

For technical support
Please contact your supplier

For further information
MicroProcessor Engineering Limited
133 Hill Lane
Southampton SO15 5AF
UK

Tel: +44 (0)23 8063 1441
Fax: +44 (0)23 8033 9691
e-mail: mpe@mpeforth.com
tech-support@mpeforth.com
web: www.mpeforth.com

Table of Contents

1 Licensing and other matters

1.1 Distribution of application programs

There are several ways in which VFX Forth applications can be distributed. These are:

- Sealed turnkey application with no access to the interactive Forth.
- Sealed except for engineering and maintenance access by the developer.
- Open Forth interpreter/compiler provided for the end user.

1.1.1 Sealed turnkey applications

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

1.1.2 Engineering and maintenance access

If the developing organisation wishes to provide what the user sees as a sealed turnkey application, but in which an open Forth can be exposed for engineering and maintenance access by the developer organisation no royalty will be charged. However a license agreement must be signed with MPE in order to protect MPE's copyright. If the company responsible for maintenance is not the developer then the maintenance company must have a license.

1.1.3 User open Forth interpreter

In order to distribute a system with an open Forth interpreter for the end user, a license agreement and royalty terms must be agreed with MPE. MPE is able to help you supply selected portions of the development environment, or to provide end user documentation. The cost of such licenses will depend on the facilities required.

1.2 Distribution of files

Unless special license terms say otherwise, this section applies.

Shipped applications may be based on the file *vfxlin*, *vfxlin386* and any number of overlays. Object code generated from the source files can of course be included in your applications. MPE source files and all other files including editors, support programs and shared libraries are part of the development environment, which may not be distributed without prior permission in writing from MicroProcessor Engineering. However, the INI parser libraries, *mpeparser.dll* or *libmpeparser.** may be distributed with your applications - these files are distrubuted under an MIT license.

The source directories provided with VFX Forth may not be distributed, and remain the intellectual property of MicroProcessor Engineering Ltd. Some source directories, e.g. the INI parser, contain additional licenses which apply to those directories only.

1.3 Evaluation and Lite compilers

These terms apply to the compilers supplied free of charge as the evaluation or Lite editions.

Commercial use of evaluation or Lite compilers is not permitted. If you sell an application written with VFX Forth, that is commercial use. If you are paid to write software with VFX Forth, that is commercial use. If you are a teacher and want to use VFX Forth in a class, that is commercial use, so contact us and we will give you written permission.

If you think that you are a special case, please contact us and we will consider your case.

1.4 Warranties, support, and copyright

We try to make VFX Forth as reliable and bug free as we possibly can. We support our products. If you find a bug in VFX Forth or its associated programs we will do our best to fix it. Please send us a disc with a piece of sample code and a paper listing of the problem, and let us know the serial number of your issue disc. We will then send you an updated disc when we have fixed the problem. Do however, contact us by fax or your supplier first in case the problem has already been fixed. Please note that the level of Technical Support that we can offer will depend on the Support Policy purchased with VFX Forth. Technical support is only provided for the current shipping version of VFX Forth.

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

As we sell copies of VFX Forth through dealers and purchasing departments we cannot keep track of all our users. If you fill out the registration form enclosed and send it back to us, we will put you on our mailing list. This way we will be able to keep you informed of updates and new extensions for VFX Forth. If you want direct technical support from us we will need these details to respond to you. You will find the serial number of the system on the invoice, original CD or memory stick, or on the email notification.

2 Installation and introduction

2.1 Installation

2.1.1 VFX Forth

Install the appropriate package for your Linux distribution and follow the on-screen instructions. If a key is required, it is a 12 digit number which you will find

- on the invoice for the Standard, Professional or Mission versions,
- the email from MPE for FTP downloads,
- the email response for the evaluation version.

The key **must** be entered as a 12 digit number with no punctuation.

2.1.2 Directory structure

The main installed directory (folder) structure looks like this:

```
/usr/bin  - issue binaries
<vfx>
  Doc
    AnsForth.Htm - ANS Forth HTML documentation
    VfxLin.htm   - VFX Forth HTML documentation
  Bin     - development binaries (not all versions)
  Examples - Examples to look at and use
    - has subdirectories
  Lib      - library of tools maintained by MPE
    CIAO  - C Inspired Active Objects OOP package
    FSL   - A port of the Forth Scientific Library
    GENIO - Examples of Generic I/O drivers
    OOP   - A Neon-style OOP package
    Lin32 - Linux specific code
  Sources  - source code if applicable
    - has subdirectories
  Kernel   - source code if applicable
    - has subdirectories
  VFXBase  - source code if applicable
    - has subdirectories
  TOOLS    - useful third party O/S specific tools
           - not present in all versions
  XTRA     - Additional third party O/S specific tools
           - not present in all versions
```

2.1.3 Executable files

The executables are in the folder <*Vfx*>*Bin* in some versions. The installation system also places the executables in */usr/bin*.

- *vfxlin* - the development version with a full set of tools. Requires a Pentium Pro or above.

- *vfxkern.elf* - a minimal kernel produced by cross compiling. This file is not available in all versions. Requires a Pentium Pro or above.
- *vfxlin386* - the development version with a full set of tools. This version was compiled for a plain 386 CPU and so can be used with lightweight hardware and distributions for embedded systems such as the Ebox2300.
- *vfxkern386.elf* - a minimal kernel produced by cross compiling. This file is not available in all versions. This version was compiled for a plain 386 CPU.
- *vfxsupp.so* - a support library for development use.
- *libmpeparser.so* - a support library for use with initialisation files.

The support library, *vfxsupp.so*, extends the number handler to search for Linux constants that are then treated as if their value had been entered numerically. The search is case sensitive, and yes we have seen constants which differ only in case and return different values. This library avoids having to keep a huge number of constant definitions in the Forth dictionary. The support library is only needed for development, and may not be shipped with applications.

2.1.4 Supported distributions

There is a tarball for self-installation.

Installing from the tarball

The tarball was created with

```
tar cvfz ...
```

Extract the tarball to a suitable directory.

```
tar xvfz ...
```

Step into the new directory and there will be a file *InstallMe.Lin32.sh*. This is a shell script which you you should run to perform all the donkey work described below.

The executables and shared libraries are in the *Bin* subdirectory. You should now copy these to the "right" places for your distribution, usually */usr/bin* and */usr/lib* for 32 bit distributions. The files to copy to */usr/bin* are *vfxlin* and/or *vfxlin386*. The files to copy to */usr/lib* are *libmpeparser.so.0* and *vfxsupp.so.1.0.1*. This last needs a symbolic link as *vfxsupp.so.1*.

```
ln -s vfxsupp.so.1.0.1 vfxsupp.so.1
```

The shared library *libmpeparser.so.0* contains the code to support persistent INI files and and *vfxsupp.so.1.0.1* contains a large number of (case-sensitive) constants from Linux header files.

2.1.5 Other flavours of Unix

We can provide ports of VFX Forth for Linux to operating systems such as BSD and Open Solaris. Porting and supporting these versions depends on "ample spare time", consultancy requests, and the number of operating systems we can cram onto the development machines.

2.2 Introduction

2.2.1 Getting started

If you do not know Forth, versions supplied on CD contain a PDF version of "Programming Forth" by Stephen Pelc. Users of evaluation versions can obtain the same PDF file from the MPE website.

Books on Forth are available from MPE and others. For more details see:

```
http://www.mpeforth.com/books.htm
```

Do not be afraid to play. Forth is an interactive system designed to help you explore its own programming environment. There is plenty of source code in the *Examples* and *Lib* directories, so look at it, edit it, and see what happens!

2.2.2 Set up your editor

VFX Forth for Linux is not supplied with an editor. If you want to set one, use:

```
editor-is <editor>
```

e.g.

```
editor-is emacs
editor-is /bin/vi
```

`SetLocate` tells VFX Forth how your editor can be called to go a particular file and line. Use in the form:

```
SetLocate  <rest of line>
```

where the text after `SetLocate` is used to define how parameters are passed to the editor, e.g. for Emacs, use:

```
SetLocate +%l% "%f%"
```

```
EMACS        +%l% "%f%"
             --no-wait +%l% "%f%"
Kate         --use --line %l% "%f%" &
UltraEdit    -- "%f%" --lc%l%:1 &
```

Thanks to Charles Curley for the additional EMACs information. See http://www.charlescurley.com. He also notes that you should add the following to your .emacs file:

```
(if (or (string-equal system-type "gnu/linux")
        (string-equal system-type "cygwin"))
    (server-start)
         (message "emacsserver started."))
```

It is essential to place the quote marks around the %f% macro if your source paths include spaces.

For `LOCATE` and other source code tools to work, VFX Forth must know where its source code is. The root of the source code directory is saved in the `VFXPATH` text macro, which is expanded when needed. To see what the current setting is, use:

```
ShowMacros
```

To see how the macro is used, look at the source file list:

```
.Sources
```

To set the macro, use something like

```
 c" ~/VfxForth/Sources" setMacro VfxPath
```

For more information on text macros, see the chapter on "Text macro substitution".

2.2.3 Set up the PDF help system

For words for which you do not have the source, but are documented in the manual, you can use `HELP <name>`, e.g.

```
help help
```

This needs configuration according to which PDF viewer and version you are using. The default incantations are for *xpdf*.

```
s" xpdf %h%.pdf %p% &" HelpCmd$ place
s" %LOAD_PATH%/../doc/VfxLin" HelpBase$ place
#17 HelpPage0 !
```

The first line tells the system how to run a PDF viewer so that it displays the page *%p%*. The second line defines the PDF file base name, which is used to find the PDF file and its associated index file. The third line defines the page offset from the start of the PDF file to page number 1, i.e. to step over the table of contents and so on.

The configuration information is preserved between sessions in a configuration file, by default *~/.VfxForth.ini*.

2.2.4 New features in this version

Changes between versions are documented in reverse chronological order in the file *<VFX>/Doc/Release.vfx.txt*, which is available from the Help menu in the Windows version. The file contains the changes for all versions.

3 How Forth is documented

The Forth words in this manual are documented using a methodology based on that used for the ANS standard document. As this is not a standards document but a user manual, we have taken some liberties to make the text easier to read. We are not always as strict with our own in-house rules as we should be. If you find an error, have a complaint about the documentation or suggestions for improvement, please send us an email or contact us in some other way.

When you browse the words in the Forth dictinary using `WORDS` or when reading source code you may come across some words which are not documented. These words are undocumented because they are words which are only used in passing as part of other words (factors), or because these words may change or may not exist in later versions.

"*Documentation is like sex: when it is good, it is very, very good; and when it is bad, it is better than nothing.*" - *Dick Brandon*

3.1 Forth words

Word names in the text are capitalised or shown in a bold fixed-point font, e.g. SWAP or `SWAP`. Forth program examples are shown in a Courier font thus:

```
: NEW-WORD   \ a b -- a b
  OVER DROP
;
```

If you see a word of the form `<name>` it usually means that `name` is a placeholder for a name you will provide.

The notation for the glossary entries in this manual have two major parts:

- The definition line.
- The description.

The definition line varies depending on the definition type. For instance - a normal Forth word will look like:

```
: and           \ n1 n2 -- n3                     6.1.0720
```

The left most column describes the word `NAME` and type (colon) the center column describes the stack effect of the word and the far right column (if it exists) will specify either the ANS language reference number or an MPE reference to distinguish between ANS standard and MPE extension words.

The stack effect may be followed by an informal comment separated from the stack effect by a ';' character.

```
: and           \ x1 x2 -- x3 ; bitwise and
```

This is a "quick reference" comment.

When you read MPE source code, you will see that most words are written in the style:

```
: foo       \ n1 n2 -- n3
\ *G This is the first glossary description line.
\ ** These are following glossary description lines.
  ...
;
```

Most MPE manuals are now written using the DocGen literate programming tool available and documented with all VFX Forths for Windows, Mac OS X and Linux. DocGen extracts documentation lines (ones that start `"\ *X "`) from the source code and produces HTML or PDF manuals.

3.2 Stack notation

```
before -- after
```

where *before* means the stack parameters before execution and *after* means stack parameters after execution. In this notation, the top of the stack is to the right. Words may also be shown in context when appropriate. Unless otherwise noted, all stack notations describe the action of the word at execution time. If it applies at compile time, the stack action is preceded by `C:` or followed by `(compiling)`

An action on the return stack whill be shown

```
R: before -- after
```

Similarly, actions on the separate float stack are marked by `F:` and on an exception stack by `E:`. The definition of `>R` would have the stack notation

```
x -- ; R: -- x
```

Defining words such as `VARIABLE` usually indicate the stack action of the defining word (`VARIABLE`) itself and the stack action of the child word. This is indicated by two stack actions separated by a ';' character, where the second action is that of the child word.

```
: VARIABLE    \ -- ; -- addr
```

In cases where confusion may occur, you may also see the following notation:

```
: VARIABLE    \ -- ; -- addr [child]
```

Unless otherwise stated all references to numbers apply to native signed integers. These will be 32 bits on 32 bit CPUs and 16 bits on embedded Forths for 8 and 16 bit CPUs. The implied range of values is shown as {from..to}. Braces show the content of an address, particularly for the contents of variables, e.g., BASE {2..72}.

The native size of an item on the Forth stack is referred to as a `CELL`. This is a 32 bit item on a 32 bit Forth, and on a byte-addressed CPU (the vast majority, most DSP chips excluded) this is a four-byte item. On many CPUs, these must be stored in memory on a four-byte address boundary for hardware or performance reasons. On 16 bit systems this is a two-byte item, and may also be aligned.

The following are the stack parameter abbreviations and types of numbers used in the documentation for 32 bit systems. On 16 bit systems the generic types will have a 16 bit range. These abbreviations may be suffixed with a digit to differentiate multiple parameters of the same type.

```
Stack         Number     Range              Field
Abbreviation  Type       (Decimal)          (Bits)
flag          boolean    0=false, nz=true   32
true          boolean    -1 (as a result)   32
false         boolean    0                  32
char          character  {0..255}           8
b             byte       {0..255}           8
w             word       {0..65535}         16
  here word means a 16 bit item, not a Forth word
n             number     {-2,147,483,648    32
                         ..2,147,483,647
x             32 bits    N/A                32
+n            +ve int    {0..2,147,483,647} 32
u             unsigned   {0..4,294,967,295} 32
addr          address    {0..4,294,967,295} 32
a-addr        address    {0..4,294,967,295} 32
  the address is aligned to a CELL boundary
c-addr        address    {0..4,294,967,295} 32
  the address is aligned to a character boundary
32b           32 bits    not applicable     32
d             signed     {-9.2e18..9.2e18}  64
              double
+d            positive   {0..9.2e18}        64
              double
ud            unsigned   {0..1.8e19}        64
              double
sys    0, 1, or more system dependent entries
char          character  {0..255}          8
"text"  text read from the input stream
```

Any other symbol refers to an arbitrary signed 32-bit integer unless otherwise noted. Because of the use of two's complement arithmetic, the signed 32-bit number (n) -1 has the same bit representation as the unsigned number (u) 4,294,967,295. Both of these numbers are within the set of unspecified weighted numbers. On many occasions where the context is obvious, informal names are used to make the documentation easier to understand.

3.3 Input text

Some Forth words read text from the input stream (e.g the keyboard or a file). That text is read from the input stream is indicated by the identifiers "<*name*>" or "*text*". This notation refers to text from the input stream, not to values on the data stack.

Likewise, *ccc* indicates a sequence of arbitrary characters accepted from the input stream until the first occurrence of the specified delimiter character. The delimiter is accepted from the input stream, but it is not one of the characters *ccc* and is therefore not otherwise processed. This notation refers to text from the input stream, not to values on the data stack.

Unless noted otherwise, the number of characters accepted may be from 0 to 255.

3.4 Other markers

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

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

`I` The word is immediate. It will be executed even during compilation, unless special action is taken, e.g. by preceding it word with the word `POSTPONE`.

`M` Affected by multi-tasking

`U` A user variable.

4 Base Kernel Definitions

This section describes a number of the base kernel definitions available to the system. This wordset includes the vast bulk of the ANS Forth specified words as well as a number of useful additions. Note that further information about some words may be found in the draft ANS specification, accessible from the Help menu.

4.1 Glossary Notation

The notation for the glossary definitions found in this manual have two major parts:

- The definition Line.
- The description Line.

The definition line varies depending on the definition type. For instance - a normal Forth word will look like:

```
: AND               \ n1 n2 -- n3                              6.1.0720
```

where the left most column describes the word `AND` and type (colon), the center column describes the stack effect of the word and the far right column will specify the ANS standard's reference ID, an MPE reference ID, `Forth200x` to indicate that the word is a standards proposal, or this field may be empty.

4.2 Main Vocabularies

`vocabulary FORTH         \ --`

The standard general purpose vocabulary.

`vocabulary ROOT          \ --`

This vocabulary contains only the words which ensure that you can select other vocabularies.

`vocabulary SYSTEM        \ --`

A repository for those words which are required internally by the compiler/system but should never appear in user code. `SYSTEM` words may be changed without notice.

`vocabulary ENVIRONMENT  \ --`

Storage for ANS ENVIRONMENT stuff.

`vocabulary SourceFiles  \ --`

Storage for SourceFile descriptions after `INCLUDE`.

`vocabulary substitutions        \ --`

Repository for text macros.

`vocabulary Externals    \ --`

Repository for external library calls.

4.3 ASCII Character Constants

Various constants for ASCII characters to aid readability and to provide some insulation between VFX Forth implementations on different operating systems.

`$07 constant ABELL      \ -- char`

Bell/sound character

`$08 constant BSIN        \ -- char`
Backspace on input character

`$7F constant DELIN       \  -- char`
Delete character

`$08 constant BSOUT       \ -- char`
Backspace on output character

`$09 constant ATAB        \ -- char`
Tab character

`$0D constant ACR         \ -- char`
Carriage Return character

`$0A constant ALF         \ -- char`
Line Feed character

`$0C constant FFEED       \ -- char`
Form Feed character

`$20 constant ABL         \ -- char`
Space character

`$2E constant ADOT        \ -- char`
Dot character

`$00 constant AEOL        \ -- char`
Generic EOL marker.

`#13 constant ANL         \ -- char`
Host specific constant for the character returned when you press the Enter key on your keyboard.

`create eol$      \ -- addr`
A counted and zero terminated string holding the operating system specific end of line sequence as a counted and zero terminated string.

- For Windows, DOS and bare metal systems without an operating system, this is CR/LF,
- For Unix and derivatives such as Linux and Mac OS X, this is LF,
- For Mac OS up to 9, this is CR.

`create crlf$     \ -- addr`
A counted and zero terminated string holding a CR/LF pair.

4.4 System CONSTANTs

Various constants for the internal system.

`0 constant FALSE                \ -- 0`   6.2.1485
The well formed flag version for a logical negative.

`-1 constant TRUE                \ -- -1`   6.2.2298
The well formed flag version for a logical positive.

`ABL constant BL                 \ -- u`   6.1.0770
An internal constant for blank space.

`$40 constant C/L                \ -- u`

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

```
64 constant #VOCS               \ -- u
```
Maximum number of Vocabularies in search order.

```
#VOCS cells constant VSIZE      \ -- u
```
Size of CONTEXT area for search order.

```
$200 constant FILETIBSZ         \ -- len
```
Size of TIB buffer when SOURCE-ID is a file pointer.

```
#260 constant MAX_PATH          \ -- len
```
Size of longest file/path name for Windows and DOS. 1024 is used for Linux and OS X.

```
$00 constant NULL
```
NULL pointer.

4.5 Defined USER Variables

USER variables are the Forth equivalent of Thread Local Storage. They are for task specific information and act as normal variables within their thread scope.

USER variables can be defined by the words USER and +USER. They are defined using an offset from a base address assigned at the start of each task.* Offsets in the USER area below $1000 are reserved for kernel use. The variable NEXTUSER is used by +USER and is initialised to $1000 in the primary build of VFX Forth, with 4k bytes of memory available for application use.

The following USER variables have been declared within the system.

```
$00 user S0             \ -- addr
```
Initial Base of data stack.

```
$04 user R0             \ -- addr
```
Initial Base of return stack.

```
$08 user #TIB           \ -- addr                              6.2.0060
```
Number of characters currently in TIB.

```
$10 user >IN            \ -- addr                              6.1.0560
```
Pointer to next char in input stream.

```
$14 user OUT            \ -- addr
```
Number of characters output since last CR.

```
$18 user BASE           \ -- addr                              6.1.0750
```
Numeric Conversion Base.

```
$1C user HLD            \ -- addr
```
Used during number formatting to point to next character to save.

```
$20 user #L             \ -- addr
```
Number of cells converted by NUMBER?.

```
$24 user #D             \ -- addr
```
Number of digits converted by NUMBER?.

```
$28 user DPL            \ -- addr
```
Position of double number indicator in number text.

`$2C user 'TIB           \ -- addr`
Address of `TIB`.

`$38 user OP-HANDLE      \ -- addr`
Generic IO output handler structure.

`$3C user IP-HANDLE      \ -- addr`
Generic IO input handler structure.

`$40 user CURROBJ        \ -- addr`
Current Object Pointer for OOP extensions.

`$44 user 'AbortText     \ -- addr`
Pointer to counted string for last `ABORT"`.

`$48 user $S0            \ -- addr`
Initial Base of string stack.

`$4C user $SP            \ -- addr`
Current string stack pointer.

`fs0-offset user fs0`
Initial Base of float stack.

`fsp-offset user fsp`
Current float stack pointer.

`$58 user line#                 \ SFP001`
Current source input line number. Note that this variable does NOT describe the number of lines output, but is reserved to hold the number of lines read from the current source input device. For console devices, `LINE#` should be set to `-1` to indicate that the source cannot be recovered for words such as `LOCATE` and `XREF`.

`$5C user op-line#              \ SFP001`
Current output line number, incremented by CR and reset by `QUIT`.

`  user PAD               \ -- addr ;                           6.2.2000`
Transient data area. The size of `PAD` is given by the constant `/PAD` in the `ENVIRONMENT` vocabulary.

4.6 System Variables and Buffers

4.6.1 Variables

`variable c/Line         \ -- addr`
Maximum number of chars/line in interpret console.

`variable c/Cols         \ -- addr`
Character height in `DEFAULT-CONSOLE` device.

`variable dp-char        \ -- addr`
Holds up to four ASCII values of double number separators. Unused bytes must be set to zero.

`variable fp-char        \ -- addr`
Holds up to four ASCII values of floating point number separators.

`variable ign-char       \ -- addr`
Holds ASCII values of characters that are ignored during number scanning. Set to ':' by default.

`variable dir1-char      \ -- addr`

CELL, holds the primary directory separator character used when scanning file names. Set to '\' by default for Windows/DOS and to '/' for Unix derivatives.

variable dir2-char \ -- addr

CELL, holds the secondary directory separator character used when scanning file names. Set to '/' by default for Windows/DOS and to '\' for Unix derivatives.

variable FENCE \ -- addr

End of protected dictionary.

variable VOC-LINK \ -- addr

Links vocabularies.

variable wid-link \ -- addr

Links word-lists.

variable res-link \ -- addr

Links resources.

variable lib-link \ -- addr

Links dynamic/shared libraries.

variable ovl-link \ -- addr

Links active overlays.

variable ovl-id \ -- addr

Holds unique overlay ID

variable <id> \ -- addr

A variable that holds the next available ID number. See NEXTID: in the Resources Section.

variable import-func-link

Links imported API functions in shared libraries.

variable SCR \ -- addr

For mass storage by old-timers.

variable BLK \ -- addr

User input device: 0 for keyboard/file, non-zero is block number.

variable STATE \ -- addr

Interpreting (0) or compiling (non-zero).

variable CSP

Stack pointer saved for error checking.

variable CURRENT \ -- addr

Holds the wordlist/vocabulary in which new definitions are created.

vsize buffer: CONTEXT \ -- addr

Search order array.

vsize buffer: MinContext \ -- addr

A CONTEXT array for minimum search order.

variable LAST \ -- addr

Points to last definition (after Link Field).

variable #THREADS \ -- addr

Default number of threads in a new wordlist.

```
variable CHECKING       \ -- addr
```
True if checking structure definitions is enabled. Note that this variable may be removed in a future release.

```
2variable SOURCE-LINE-POS       \ -- addr
```
Contains double file position before refill.

```
variable Saved>IN       \ -- addr
```
Holds the value of `>IN` before each token parse in interpret.

```
variable <HeaderLess>   \ -- addr
```
A flag. Declares the presence of a header in the last definition.

```
variable 'SourceFile    \ -- addr
```
Pointer to source include struct for current file, or 0.

```
variable tabwordstop    \ -- addr
```
Cursor X Position for tab stops.

```
variable Optimising     \ -- addr
```
Variable is set TRUE when optimisation should be used.

```
variable NextUser
```
Next Valid offset for a new user variable.

```
variable OPERATORTYPE   \ -- addr
```
Set by prefix operators such as `TO` and `ADDR`.

```
variable Top-Mask       \ -- addr ; controls loop alignment
```
Mask that controls the alignment of loop heads during code generation.

```
variable TextChain      \ -- addr
```
Anchor for the linked list of error message structures.

```
variable debug1         \ -- addr
```
When set, `INCLUDE` displays the lines of the file.

4.6.2 Values

```
0 value FpSystem        \ -- n
```
The value `FPSYSTEM` defines which floating point pack is installed and active for compilation. See the Floating Point chapter for further details.

```
0 value original-xt     \ -- xt
```
Set during a redefinition to preserve the xt of the word being redefined.

4.7 Kernel DEFERred words

These words are `DEFER`red to allow later modification.

4.7.1 Input and Output

Although the standard Forth I/O functions are deferred, users are strongly encouraged to use the generic I/O mechanism rather than to change the global effect of the I/O words. The I/O words are `DEFER`red for historical reasons and to ease porting.

```
defer EMIT      \ char -- ; display char
```
Display char on the current I/O device.

`defer EMIT?      \ -- ior`
Return a non-zero ior if the current output device is ready to receive a character. The ior may be device dependent.

`defer KEY        \ -- char ; receive char`
Wait until the current input device receives a character and return it.

`defer KEY?       \ -- flag ; check receive char`
Return true if a character is available at the current input device.

`defer EKEY       \ -- char ; receive char`
Wait until the current input device receives a character and return it. Note that the behaviour of `EKEY` and `EKEY?` may be implementation dependent. See the ANS Forth standard for more details.

`defer EKEY?       \ -- flag ; check receive char`
Return true if a character is available at the current input device. Note that the behaviour of `EKEY` and `EKEY?` may be implementation dependent. See the ANS Forth standard for more details.

`defer CR         \ -- ; display new line`
Perform the equivalent of a CR/LF pair on the current output device. This action may be device dependent.

`defer TYPE       \ c-addr len -- ; display string`
Display/write the string on the current output device.

`defer ACCEPT     \ c-addr +n1 -- +n2`
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 a CR. The action may be input device specific.

4.7.2 Kernel and Convenience

These words are deferred to improve kernel portability, and to provide points at which the default behaviour of the Forth kernel can be changed.

`defer EntryPoint        \ hmodule 0 commandline show -- res`
This word is the entry point from the startup code to the Forth system. The arguments follow the WinMain conventions, except that the command line may include the program name. See the chapter about creating turnkey applications for more and important details.

`defer ABORT      \ i*x -- ; R: j*x -- ; error handler`
Empty the data stack and perform the action of `QUIT`, which includes emptying the return stack, without displaying a message.

`defer NUMBER?    \ addr -- d/n/- 2/1/0`
Attempt to convert the counted string at 'addr' to an integer. The return result is either zero for failed, a single cell number and one for a single-cell conversion, or a double cell number and two for a double number conversion. The ASCII number string supplied can also contain an explicit radix (number base) override. A leading $ enforces hexadecimal, a leading # enforces decimal and a leading % enforces binary. Hexadecimal numbers can also be specified by a leading '0x' or trailing 'h'. After a floating point pack has been compiled from the LIB directory, the action of `NUMBER?` is changed to accept floating point numbers as well as integers.

`defer ShowSourceOnErrorHook      \ --`
Performed at the end of `SHOWSOURCEONERROR`.

`defer EditOnError        \ --`

Performed in `DOERRORMESSAGE`. The default is `NOOP`. This word is assigned a new action by the Studio environment.

`defer pause      \ --`

The multitasker is installed here. Until a multitasker is installed the action is `NOOP` or `YIELD`. Do **not** call `PAUSE` inside callbacks.

`defer ms         \ n --`

Wait for n milliseconds.

`defer ticks      \ -- n`

Return the system timer value in milliseconds. Treat the returned value as a 32 bit unsigned number that wraps on overflow.

`defer interpret \ i*x -- j*x ; process current input line`

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

`defer QUIT               \ -- ; R: i*x --                    6.1.2050`

Empty the return stack, store 0 in `SOURCE-ID`, make the console the current input device, and enter interpretation state. `QUIT` repeatedly `ACCEPT`s a line of input and `INTERPRET`s it, with a prompt if in interpretation state. See the separate chapters on error handling and internationalisation for details of error message display.

`defer .Prompt    \ --`

The Forth console prompt.

4.7.3 GUI interface hooks

These words provide hooks into systems, both GUI and kernel, which use message passing or event handlers. These words are mostly used by Generic I/O devices while waiting.

`defer Idle               \ --`

Windows only: Despatches the next message, waiting if none are present. `Idle` only returns when a message has been received.

`defer WaitIdle           \ --`

Linux, OS X and DOS: Despatches the next message/event, waiting if none are present. `WaitIdle` only returns when a message has been received.

`defer BusyIdle           \ --`

Despatches a message/event if available. The word returns immediately if no messages are available. The default action is `(BusyIdle)`. See also `EmptyIdle`.

`defer EmptyIdle \ --`

Empty the message/event loop. This can be used inside applications to ensure that a GUI system has an opportunity to process messages/events.

4.8 Logic functions

Perform various logic and bit based operations on stack items.

`: and            \ n1 n2 -- n3                            6.1.0720`

Perform a logical AND between the top two stack items and retain the result in top of stack.

`: or             \ n1 n2 -- n3                            6.1.1980`

Perform a logical OR between the top two stack items and retain the result in top of stack.

`: xor            \ n1 n2 -- n3                                    6.1.2490`
Perform a logical XOR between the top two stack items and retain the result in top of stack.

`: not            \ n1 -- n1`
Perform a bitwise NOT on the top stack item and retain result. OBSOLETE from nearly twenty years ago. `NOT` will be removed very soon. Replace this word with `INVERT`. If `NOT` is used before a conditional branch such as `IF` you may get better code by replacing `NOT` with `0=`.

`: invert         \ n1 -- ~n1                                      6.1.1720`
Perform a bitwise inversion.

`: and!           \ x addr --`
Logical AND x into the cell at addr.

`: or!            \ x addr --`
Logical OR x into the cell at addr.

`: xor!            \ x addr --`
Logical XOR x into the cell at addr.

`: bic!            \ x addr --`
Invert x and logical AND it into the cell at addr. The effect is to clear the bits at addr that are set in x.

`: false=         \ n1 -- flag`
Perform a logical NOT on the top stack item.

4.9 Stack manipulations

The following words manipulate items on the data and return stacks

`: NOOP           \ --`
A NOOP, null instruction.

`: NIP            \ x1 x2 -- x2                                    6.2.1930`
Dispose of the second item on the data stack.

`: TUCK           \ x1 x2 -- x2 x1 x2                              6.2.2300`
Insert a copy of the top data stack item underneath the current second item.

`: PICK           \ xu .. x0 u -- xu .. x0 xu                      6.2.2030`
Get a copy of the Nth data stack item and place on top of stack. `0 PICK` is equivalent to `DUP`.

`: RPICK          \ n -- a`
Get a copy of the Nth return stack item and place on top of stack.

`: ROLL           \ nn..n0 n -- nn-1..n0 nn                        6.2.2150`
Rotate the order of the top N stack items by one place such that the current top of stack becomes the second item and the Nth item becomes TOS. See also `ROT`.

`: nrev           \ xN...x1 count -- x1....xN`
Reverse the order of the top N data stack items.

`: nDrop          \ XN..X1 N -- xn..x2`
Drop N items from the data stack.

`: ROT            \ n1 n2 n3 -- n2 n3 n1                           6.1.2160`
ROTate the positions of the top three stack items such that the current top of stack becomes the second item. See also `ROLL`.

`: -ROT \ n1 n2 n3 -- n3 n1 n2`
The inverse of `ROT`.

`: >R \ x -- ; R: -- x` 6.1.0580
Push the current top item of the data stack onto the top of the return stack.

`: R> \ -- x ; R: x --` 6.1.2060
Pop the top item off the return stack and place on the data stack.

`: R@ \ -- x` 6.1.2070
Copy the top item of the return stack and place on the data stack.

`: 2>R \ x1 x2 -- ; R: -- x1 x2` 6.2.0340
Transfer the two top data stack items to the return stack.

`: 2R> \ -- x1 x2 ; R: x1 x2 --` 6.2.0410
Transfer the top two return stack items to the data stack.

`: 2R@ \ -- x1 x2 ; R: x1 x2 -- x1 x2` 6.2.0415
Copy the top two return stack items to the data stack.

`: 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.

`: DROP \ x --` 6.1.1260
Lose the top data stack item and promote NOS to TOS.

`: 2DROP \ x1 x2 --` 6.1.1290
Discard the top two data stack items.

`: 3drop \ x1 x2 x3 --`
Discard the top three data stack items.

`: 4drop \ x1 x2 x3 x4 --`
Discard the top four data stack items.

`: SWAP \ x1 x2 -- x2 x1` 6.1.2260
Exchange the top two data stack items.

`: 2SWAP \ x1 x2 x3 x4 -- x3 x4 x1 x2` 6.1.0430
Exchange the top two cell-pairs on the data stack.

`: DUP \ x -- x x` 6.1.1290
Duplicate the top stack item.

`: ?DUP \ x -- 0 | x x` 6.1.0630
`DUP`licate the top stack item only if it is non-zero.

`: 2rot \ 1 2 3 4 5 6 -- 3 4 5 6 1 2` 8.6.2.0420
Perform `ROT` operation on 3 double numbers.

`: 2DUP \ x1 x2 -- x1 x2 x1 x2` 6.1.0380
`DUP`licate the top cell-pair on the data stack.

`: 3dup \ x1 x2 x3 -- x1 x2 x3 x1 x2 x3`
`DUP`licate the top three items on the data stack.

`: 4dup \ x1 x2 x3 x4 -- x1 x2 x3 x4 x1 x2 x3 x4`

DUPlicate the top 4 data stack items.

```
: OVER          \ x1 x2 -- x1 x2 x1                          6.1.1990
```
Copy NOS to a new top-of-stack item.

```
: 2OVER         \ x1 x2 x3 x4 -- x1 x2 x3 x4 x1 x2           6.1.0400
```
Similar to `OVER` but works with cell-pairs rather than cell items.

```
: UP@           \ -- up
```
Get the current address value of the user-area pointer.

```
: UP!           \ up --
```
Set the current address value of the user-area pointer.

```
: SP@           \ -- n
```
Get the current address value of the data-stack pointer.

```
: SP!           \ n --
```
Set the current address value of the data-stack pointer.

```
: RP@           \ -- m
```
Get the current address value of the return-stack pointer.

```
: RP!           \ m --
```
Set the current address value of the return-stack pointer.

```
: DEPTH         \ -- +n                                      6.1.1200
```
Return the number of items on the data stack, excluding the count.

```
: RDEPTH        \ -- +n
```
Return the number of items on the return stack.

```
: min           \ n1 n2 -- n1|n2                             6.1.1880
```
Given two data stack items preserve only the smallest.

```
: MAX           \ n1 n2 -- n1|n2                             6.1.1870
```
Given two data stack items preserve only the largest.

```
: umin           \ n1 n2 -- n1|n2
```
Given two data stack items preserve only the smallest.

```
: umax           \ n1 n2 -- n1|n2
```
Given two data stack items preserve only the largest.

```
: LOWORD        \ n -- n16
```
Mask off the low 16 bits of a cell.

```
: HIWORD        \ n -- n16
```
Mask off the high 16 bits of a cell and shift right by 16 bits.

```
: MAKELONG      \ lo hi -- 32bit
```
Given two 16 bit numbers produce a single 32 bit one.

```
: nswWiden      \ ... n --
```
Do a signed extension from 16 to 32 bit on Nth stack item.

```
: nsbWiden      \ ... n --
```
Do a signed extension from 8 to 32 bit on Nth stack item.

```
: nuwWiden      \ ... n --
```
Do an unsigned extension from 16 to 32 bit on Nth stack item.

`: nubWiden        \ ... n --`
Do an unsigned extension from 8 to 32 bit on Nth stack item.

4.10 Comparisons

Various words to compare stack items and return flags.

`: 0=              \ n -- t/f                              6.1.0270`
Compare the top stack item with 0 and return TRUE if equals.

`: 0<>             \ n -- t/f                              6.2.0260`
Compare the top stack item with 0 and return TRUE if not-equal.

`: 0<              \ n -- t/f                              6.1.0250`
Return TRUE if the top of stack is less-than-zero.

`: 0>              \ n -- t/f                              6.2.0280`
Return TRUE if the top of stack is greater-than-zero.

`: =               \ n1 n2 -- t/f                          6.1.0530`
Return TRUE if the two topmost stack items are equal.

`: <>              \ n1 n2 -- t/f                          6.2.0500`
Return TRUE if the two topmost stack items are different.

`: <               \ n1 n2 -- t/f                          6.1.0480`
Return TRUE if the second stack item is less than the topmost.

`: >               \ n1 n2 -- t/f                          6.1.0540`
Return TRUE if the second stack item is greater than the topmost.

`: <=              \ n1 n2 -- t/f`
Return TRUE if the second stack item is less than or equal to the topmost.

`: >=              \ n1 n2 -- t/f`
Return TRUE if the second stack item is greater than or equal to the topmost.

`: U>              \ n1 n2 -- t/f                          6.2.2350`
An UNSIGNED version of `>`.

`: U<              \ n1 n2 -- t/f                          6.1.2340`
An UNSIGNED version of `<`.

`: U>=             \ n1 n2 -- t/f`
An UNSIGNED version of `>=`.

`: U<=             \ n1 n2 -- t/f`
An UNSIGNED version of `<=`.

`: WITHIN?         \ n1 n2 n3 -- flag`
Return TRUE if N1 is within the range N2..N3. This word uses signed arithmetic.

`: WITHIN          \ n1|u1 n2|u2 n3|u3 -- flag             6.2.2440`
Return TRUE if n2|u2 <= n1|u1 < n3 The ANS version of `WITHIN?`. Note the conditions This word uses unsigned arithmetic, so that signed compares are treated as existing on a number circle.

`: D0<             \ d -- flag                           8.6.1.1075`
Return true if *d* is less than zero (is negative).

`: D0=            \ d -- flag                                   8.6.1.1080`

Return true if *d* is zero.

`: D0<>           \ d -- flag`

Return true if *d* is non-zero.

`: d=             \ d1 d2 -- flag                               8.6.1.1080`

Return true if the two double numbers are equal.

`: d<             \ d1 d2 -- flag                               8.6.1.1110`

Return TRUE if the double number d1 is less than the double number d2.

`: d>             \ d1 d2 -- flag`

Return TRUE if the double number d1 is greater than the double number d2.

`: dmax           \ d1 d2 -- d1|d2                              8.6.1.1210`

Return the maximum double number from the two supplied.

`: dmin           \ d1 d2 -- d1|d2                              8.6.1.1220`

Return the minimum double number from the two supplied.

`: DU<            \ ud1 ud2 -- flag`

True if ud1<ud2.

`: DU>            \ ud1 ud2 -- flag`

True if ud1>ud2.

4.11 Arithmetic Operators.

4.11.1 Shifts

`: LSHIFT         \ x1 u -- x2                                  6.1.1805`

Logically shift X1 by U bits left. The result of shifting by more than 31 bits is undefined.

`: RSHIFT         \ x1 u -- x2                                  6.1.2162`

Logically shift X1 by U bits right. The result of shifting by more than 31 bits is undefined.

`: arshift        \ x1 u -- x2`

Shift x1 right by u bits, filling with the previous top bit. An arithmetic right shift. The result of shifting by more than 31 bits is undefined.

`: ROL            \ x1 u -- x2`

Logically rotate X1 by U bits left. The result of shifting by more than 31 bits is undefined.

`: ROR            \ x1 u -- x2`

Logically rotate X1 by U bits right. The result of shifting by more than 31 bits is undefined.

4.11.2 Multiplication

`: *              \ n1 n2 -- n3                                 6.1.0090`

Standard signed multiply. N3 = n1 * n2.

`: M*             \ n1 n2 -- d                                  6.1.1810`

Signed multiply yielding double result.

`: UM*            \ u1 u2 -- ud                                 6.1.2360`

Perform unsigned-multiply between two numbers and return double result.

`: UM*            \ u1 u2 -- ud                                 6.1.2360`

Perform unsigned-multiply between two numbers and return double result.

```
: D2*            \ d1 -- d1*2                                    8.6.1.1090
```
Multiply the given double number by two.

4.11.3 Division

The ANS specification contains a discussion of symmetric and floored division.

Division produces a quotient q and a remainder r by dividing operand a by operand b. Division operations return q, r, or both. The identity

```
b*q + r = a
```

shall hold for all a and b.

When unsigned integers are divided and the remainder is not zero, q is the largest integer less than the true quotient.

When signed integers are divided, the remainder is not zero, and a and b have the same sign, q is the largest integer less than the true quotient. If only one operand is negative, whether q is rounded toward negative infinity (floored division) or rounded towards zero (symmetric division) is implementation defined.

Floored division is integer division in which the remainder carries the sign of the divisor or is zero, and the quotient is rounded to its arithmetic floor. Symmetric division is integer division in which the remainder carries the sign of the dividend or is zero and the quotient is the mathematical quotient rounded towards zero or truncated. Examples of each are shown in the tables below.

```
Floored Division Example

Dividend        Divisor Remainder       Quotient
--------        ------- ---------       --------
10                 7        3               1
-10                7        4              -2
10                -7       -4              -2
-10               -7       -3               1
```

```
Symmetric Division Example
Dividend        Divisor Remainder       Quotient
--------        ------- ---------       --------
10                 7        3               1
-10                7       -3              -1
10                -7        3              -1
-10               -7       -3               1
```

Unless otherwise noted or specified, VFX Forth uses symmetric division.

```
: UM/MOD         \ ud u -- urem uquot                              6.1.2370
```
Perform unsigned division of double number UD by single number U and return remainder and quotient.

```
: SM/REM         \ d1 n1 -- n2 n3                                6.1.2214
```

Divide d1 by n1, giving the symmetric quotient n3 and the remainder n2. Input and output stack arguments are signed. An ambiguous condition exists if n1 is zero or if the quotient lies outside the range of a single-cell signed integer.

`: FM/MOD          \ d1 n1 -- n2 n3                              6.1.1561`

Divide d1 by n1, giving the floored quotient n3 and the remainder n2. Input and output stack arguments are signed. An ambiguous condition exists if n1 is zero or if the quotient lies outside the range of a single-cell signed integer.

`: MU/MOD          \ ud1 u2 -- urem ud#quot`

Perform an unsigned divide of a double by a single, returning a single remainder and a double quotient.

`: /MOD            \ n1 n2 -- rem quot                            6.1.0240`

Signed division of N1 by N2 (single-precision) yielding remainder and quotient.

`: /               \ n1 n2 -- n3                                  6.1.0230`

Standard signed division operator. n3 = n1/n2.

`: u/              \ u1 u2 -- u3`

Unsigned division operator. U3 = u1/u2.

`: MOD             \ n1 n2 -- n3                                  6.1.1890`

Return remainder of division of N1 by N2. n3 = n1 mod n2.

`: M/              \ d n1 -- n2`

Signed divide of a double by a single integer.

`: D2/             \ d1 -- d1/2                                 8.6.1.1100`

Divide the given double number by two. Signed and implemented as an arithmetic right shift, and so produces floored dividion.

4.11.4 Combined multiply and divide

These words provide combined multiply and divide operations with extended precision intermediate results. The point is to prevent overflow during integer scaling operations.

`: */MOD           \ n1 n2 n3 -- rem quot                        6.1.0110`

Multiply n1 by n2 to give a double precision result, and then divide it by n3 returning the remainder and quotient. The point of this operation is to avoid loss of precision.

`: */              \ n1 n2 n3 -- n4                              6.1.0100`

Multiply n1 by n2 to give a double precision result, and then divide it by n3 returning the quotient. The point of this operation is to avoid loss of precision.

`: m*/             \ d1 n2 +n3 -- dquot`

The result dquot=(d1*n2)/n3. The intermediate value d1*n2 is triple-precision. In an ANS Forth standard program n3 can only be a positive signed number and a negative value for n3 generates an ambiguous condition, which may cause an error on other implementations.

4.11.5 Traditional short forms

`: 1+              \ n1|u1 -- n2|u2                               6.1.0290`

Add one to top-of stack.

`: 2+              \ n1|u1 -- n2|u2`

Add two to top-of stack.

`: 4+              \ n1|u1 -- n2|u2`

Add four to top-of stack.

`: 1-             \ n1|u1 -- n2|u2                          6.1.0300`
Subtract one from top-of stack.

`: 2-             \ n1|u1 -- n2|u2`
Subtract two from top-of stack.

`: 4-             \ n1|u1 -- n2|u2`
Subtract four from top-of stack.

`: 2*             \ x1 -- x2                                6.1.0320`
Signed multiply top of stack by 2.

`: 4*             \ x1 -- x2`
Signed Multiply top of stack by 4.

`: 2/             \ x1 -- x2                                6.1.0330`
Right shift x1 one bit, sign preserved. From build 1276 onwards, this is an ANS compliant signed right shift. For an unsigned result, use `U2/` or `1 RSHIFT`.

`: U2/            \ x1 -- x2`
Unsigned divide top of stack by 2.

`: 4/             \ x1 -- x2`
Right shift x1 two bits, sign preserved. From build 1276 onwards, this is an ANS compliant signed right shift. For an unsigned result, use `U4/` below or `2 RSHIFT`.

`: u4/            \ x1 -- x2`
Unsigned divide top of stack by 4.

4.11.6 Addition and subtraction

`: +              \ n1|u1 n2|u2 -- n3|u3                    6.1.0120`
Add two single precision integer numbers.

`: -              \ n1|u1 n2|u2 -- n3|u3                    6.1.0160`
Subtract two single precision integer numbers.

`: D+             \ d1 d2 -- d3                           8.6.1.1040`
Add two double precision integers together.

`: D-             \ d1 d2 -- d3                           8.6.1.1050`
Subtract two double precision integers. D3=D1-D2.

`: M+             \ d1 n -- d2                            8.6.1.1830`
Add double d1 to sign extended single n to form double d2.

4.11.7 Negation and absolution

`: NEGATE         \ n1 -- n2                                6.1.1910`
Negate a single precision integer number.

`: ?NEGATE        \ n1 flag -- n1|n2`
If flag is negative, then negate n1.

`: ABS            \ n -- u                                  6.1.0690`
If n is negative, return its positive equivalent (absolute value).

`: DNEGATE        \ d -- -d                               8.6.1.1230`
Negate a double number.

`: ?dnegate      \ d n -- d'`
If n is negative, negate the double number d.

`: DABS          \ d -- |d|` 8.6.1.1160
Double precision version of `ABS`.

4.11.8 Converting between single and double numbers

`: S>D           \ n -- d` 6.1.2170
Convert a single number to a double one.

`: D>S           \ d -- n` 8.6.1.1140
Convert a Double number to a single.

4.11.9 Portability aids

These words make porting code between 16, 32, and 64 bit systems much easier. They avoid the use of heritage shortforms such as `2+` and `4+` which are dependent on the size of items on the data stack and in memory.

`: CELL+         \ a-addr1 -- a-addr2` 6.1.0880
Add size of a cell to the top-of stack.

`: CELLS         \ n1 -- n2` 6.1.0890
Return size in address units of N1 cells in memory.

`: CELL/         \ n1 -- n2`
Divide top stack item by the size of a cell.

`: CELLS+        \ n1 n2 -- n3`
Modify address 'n1' by the size of 'n2' cells.

`: CELL-         \ a-addr1 -- a-addr2`
Decrement an address by the size of a cell.

`: CELL          \ -- n`
Return the size in address units of one cell.

`: CHAR+         \ c-addr1 -- c-addr2` 6.1.0897
Increment an address by the size of a character.

`: CHARS         \ n1 -- n2` 6.1.0898
Return size in address units of N1 characters.

4.12 Dictionary Memory Manipulation

The following definitions provide the primitives for manipulation of dictionary memory.

`variable DP     \ -- addr`
Holds the address of the next free location in the dictionary.

`: HERE          \ -- addr` 6.1.1650
Return the current dictionary pointer which is the first address-unit of free space within the system.

`: ALLOT         \ n --` 6.1.0710
Allocate N address-units of data space from the current value of `HERE` and move the pointer.

`: ALLOT&ERASE   \ n --`

Allot n bytes of dictionary space and fill with Zero.

```
: ,             \ x --                                    6.1.0150
```
Place the CELL value X into the dictionary at `HERE` and increment the pointer.

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

```
: C,            \ x --                                    6.1.0860
```
Place the CHAR value X into the dictionary at `HERE` and increment the pointer.

```
: ALIGNED       \ addr -- a-addr                          6.1.0706
```
Given an address pointer this word will return the next `ALIGNED` address subject to system wide alignment restrictions.

```
: HALF-ALIGNED  \ addr -- a-addr
```
Align an address pointer to within half the size of a CELL.

```
: ALIGN         \ --                                      6.1.0705
```
Align dictionary pointer using the same rules as `ALIGNED`. Unused dictionary space is `ERASE`d.

```
: HALF-ALIGN    \ --
```
Align the dicionary pointer to a half-cell boundary. Unused dictionary space is `ERASE`d.)

4.13 Branch and flow control

The following definitions allow for a variety of loops and conditional execution contructs.

```
: I             \ -- n                                   6.1.1680
```
Return the current index of the inner-most `DO ... LOOP`.

```
: J             \ -- n                                   6.1.1730
```
Return the current index of the second `DO ... LOOP`.

```
: unloop        \ -- ; R: loop-sys --                    6.1.2380
```
Remove the `DO ... LOOP` control parameters from the return stack.

```
: 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.

```
: EXIT          \ -- ; R: next-sys --                    6.1.1380
```
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.

```
: EXECUTE       \ xt --                                  6.1.1370
```
Execute the code described by the XT. This is a Forth equivalent of an assembler JSR/CALL instruction.

```
: DO            \ Run: n1|u1 n2|u2 -- ; R: -- loop-sys    6.1.1240
```
Begin a `DO ... LOOP` construct. Takes the end-value and start-value from the stack.

```
: ?DO           \ Run: n1|u1 n2|u2 -- ; R: -- | loop-sys  6.2.0620
```
Compile a `DO` which will only begin loop execution if the loop parameters do not specify an interation count of 0.

```
: LOOP          \ Run: -- ; R: loop-sys1 -- | loop-sys2  6.1.1800
```
The closing statement of a `DO ... LOOP` construct. Increments the index and terminates when the index crosses the limit.

`: +LOOP          \ Run: n -- ; R: loop-sys1 -- | loop-sys2       6.1.0140`
As `LOOP` except that you specify the increment on the stack. The action of `n +LOOP` is peculiar when n is negative:

```
-1 0 ?DO  i .  -1 +LOOP
```

prints `0 -1`, whereas:

```
0 0 ?DO  i .  -1 +LOOP
```

prints nothing. This a result of the mathematical trick used to detect the terminating condition. To prevent confusion avoid using `n +LOOP` with negative *n*.

`: LEAVE          \ -- ; R: loop-sys --                          6.1.1760`
Compile code to exit a `DO ... LOOP`. Similar to 'C' language `break`.

`: ?LEAVE         \ flag -- ; R: loop-sys --`
A version of `LEAVE` which only takes effect if the given flag is non-zero.

`: BEGIN          \ C: -- dest ; Run: --                         6.1.0760`
Mark the start of a `BEGIN..[WHILE]..UNTIL/AGAIN/REPEAT` construct.

`: AGAIN          \ C: dest -- ; Run: --                         6.2.0700`
The end of a `BEGIN..AGAIN` construct which specifies an infinite loop.

`: UNTIL          \ C: dest -- ; Run: flag --                     6.1.2390`
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: flag --           6.1.2430`
Separate the condition test from the loop code in a `BEGIN..WHILE..REPEAT` block.

`: REPEAT         \ C: orig dest -- ; Run: --                    6.1.2140`
Loop back to the conditional test code in a `BEGIN..WHILE..REPEAT` construct.

`: IF             \ C: -- orig ; Run: x --                       6.1.1700`
Mark the start of an `IF ... [ELSE] ... THEN` conditional block. `ELSE` is optional.

`: THEN           \ C: orig -- ; Run: --                         6.1.2270`
Mark the end of an `IF..THEN` or `IF..ELSE..THEN` conditional block.

`: ENDIF          \ C: orig -- ; Run: --`
An alias for `THEN`. Note that ANS Forth describes `THEN` not `ENDIF`.

`: AHEAD          \ C: -- orig ; Run: --                      15.6.2.0702`
Start an unconditional forward branch which will be resolved later.

`: ELSE           \ C: orig1 -- orig2 ; Run: --                  6.1.1310`
Begin the failure condition code for an `IF`.

`: CASE           \ C: -- case-sys ; Run: --                     6.2.0873`
Begin a `CASE..ENDCASE` construct. Similar to the C `switch`.

`: OF             \ C: -- of-sys ; Run: x1 x2 -- | x1            6.2.1950`
Begin conditional block for `CASE`, executed when the switch value *x1* is equal to *x2*.

`: ?OF            \ C: -- of-sys ; Run: flag --`
Begin conditional block for `CASE`, executed when the flag is true.

`: ENDOF          \ C: case-sys1 of-sys -- case-sys2 ; Run: --    6.2.1343`

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 --                  6.2.1342
```
Terminate a `CASE ... ENDCASE` construct. `DROP`s the switch value from the stack.

```
: 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 or another result is to be returned.

```
: BEGINCASE     \ C: -- case-sys ; Run: --                    6.2.0873
```
Start a `BEGINCASE ... NEXTCASE` construct. This construct is a loop which uses `OF ... ENDOF` clauses like `CASE ... ENDCASE`, but the loop terminates after the action in the `OF ... ENDOF` clause. `BEGINCASE ... NEXTCASE` is used to construct multiple-exit loops without the appearance of spaghetti code.

```
: NEXTCASE      \ C: case-sys -- ; Run: x --
```
Terminate a `BEGINCASE ... NEXTCASE` construct. `DROP`s the switch value and branches back to `BEGINCASE`. Note that from VFX Forth 4.0, `BEGINCASE` **must** be used with `NEXTCASE`.

```
: NEXT-CASE     \ C: case-sys -- ; Run: --
```
A Version of `NEXTCASE` which does not drop the switch value. Used when the switch value itself is consumed by a default condition. Note that from VFX Forth 4.0, `BEGINCASE` **must** be used with `NEXTCASE`.

```
: cs-pick       \ xu .. x0 u -- xu .. x0 xu                15.6.2.1015
```
Get a copy of the *u*th compilation stack item.

```
: cs-roll       \ xu..x0 u -- xu-1..x0 xu                  15.6.2.1020
```
Rotate the order of the top *u* compilation stack items by one place such that the current top of stack becomes the second item and the *u*th item becomes TOS.

```
: cs-drop       \ x --
```
Discard the top item on the compilation stack.

```
: RECURSE       \ Comp: --                                    6.1.2120
```
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`.

4.14 Memory operators

The following words are used to operate on memory locations and arbitrary memory blocks.

```
: ON            \ addr --
```
Given the address of a CELL this will set its contents to TRUE.

```
: OFF           \ addr --
```
Given the address of a CELL this will set its contents to FALSE.

```
: @OFF          \ addr -- x
```
Read cell at addr, and set it to 0.

```
: @on           \ addr -- val
```
Fetch contents of addr and set to -1.

`: +!              \ n addr --                                   6.1.0130`
Add N to the CELL at memory address ADDR.

`: w+!             \ w addr --`
Add W to the 16 bit word at memory address ADDR.

`: C+!             \ b addr --`
Add B to the character (byte) at memory address ADDR.

`: -!              \ n addr --`
Subtract N from the CELL at memory address ADDR.

`: w-!             \ w addr --`
Subtract W from the 16 bit word at memory address ADDR.

`: C-!             \ b addr --`
Subtract B from the character (byte) at memory address ADDR.

`: incr            \ a-addr --`
Increment the data cell at a-addr by one.

`: decr            \ a-addr --`
Decrement the data cell at a-addr by one.

`: 2@              \ a-addr -- x1 x2                             6.1.0350`
Fetch and return the two CELLS from memory ADDR and ADDR+sizeof(CELL). The cell at the lower address is on the top of the stack.

`: @               \ addr -- n                                   6.1.0650`
Fetch and return the CELL at memory ADDR.

`: w@              \ addr -- val`
Fetch and 0 extend the word (16 bit) at memory ADDR.

`: c@              \ addr -- val                                 6.1.0870`
Fetch and 0 extend the character at memory ADDR

`: w@s             \ addr -- val(signed)`
A sign extending version of `W@`.

`: c@s             \ addr -- val(signed)`
A sign extending version of `C@`.

`: 2!              \ x1 x2 addr --                               6.1.0310`
Store the two CELLS x1 and x2 at memory ADDR. X2 is stored at ADDR and X1 is stored at ADDR+CELL.

`: !               \ n addr --                                   6.1.0010`
Store the CELL quantity N at memory ADDR.

`: w!              \ val addr --`
Store the word (16 bit) quantity VAL at memory ADDR.

`: c!              \ val addr --                                 6.1.0850`
Store the character VAL at memory ADDR.

`: fill            \ addr len char --                            6.1.1540`
Fill *LEN* bytes of memory starting at *ADDR* with the byte information specified as *CHAR*.

`: set-bit         \ mask c-addr --`
Apply the *mask* ORred with the contents of *c-addr*. Byte operation.

`: clear-bit     \ mask c-addr --`

Apply the *mask* inverted and ANDed with the contents of *c-addr*. Byte operation.

`: toggle-bit    \ mask c-addr --`

Invert the bits at *c-addr* specified by the *mask*. Byte operation.

`: test-bit      \ mask addr -- flag`

AND the *mask* with the contents of *addr* and return true if the result is non-zero (-1) or false (0) if the result is zero.

`: cmove         \ addr1 addr2 count --` 17.6.1.0910

Copy *COUNT* bytes of memory forwards from *ADDR1* to *ADDR2*. Note that as VFX Forth characters are 8 bit units, there is an implicit connection between a byte and a character.

`: cmove>        \ addr1 addr2 count --` 17.6.1.0920

As `CMOVE` but working in the opposite direction, copying the last character in the string first. Note that as VFX Forth characters are 8 bit units, there is an implicit connection between a byte and a character.

`: MOVE          \ addr1 addr2 u  --` 6.1.1900

An intelligent memory move which avoids memory overlap problems. Note that as VFX Forth characters are 8 bit units, there is an implicit connection between a byte and a character.

`: movex         \ src dest +n --`

An optimised version of `MOVE`. If n<=0, no action is taken.

`: ERASE         \ a-addr u --` 6.2.1350

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

`: BLANK         \ a-addr u --` 17.6.1.0780

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

`: UNUSED        \ -- u` 6.2.2395

Return the number of bytes free in the dictionary.

4.15 String operators

The following words are used to operate on strings. With care, some of them may also be used on arbitrary memory blocks.

In modern Forth strings are usually described by caddr/len pairs on the stack `( -- caddr len)`, where *caddr* points to first character and *len* is the number of characters in the string. Another form often used is counted strings `{ -- caddr )` in which caddr points to a count byte that is then followed by that many characters. Zero terminated strings are supported and are used for interfacing with the operating system and other libraries. Zero terminated string handling is described in a separate section of this manual.

In VFX Forth implementations for byte-addressed CPUs such as are used on PCs, a character is a byte-sized item. This means that the common assumption that a character=byte is true. However, if your code has to be ported to CPUs for which this assumption is not true (e.g. DSPs) or for which the size of a character is not one byte, then be very careful.

4.15.1 Caddr/len strings

`: /string       \ addr len n -- addr+n len-n` 17.6.1.0245

Modify a string address and length to remove the first N characters from the string.

`: SKIP          \ c-addr u char -- 'c-addr 'u`

Modify string description by skipping over leading occurrences of *char*. Note that when a space char is given, tabs are also ignored.

`: scan          \ caddr u char -- caddr2 u2`

Look for first occurrence of *char* in string and return the new string. *C-addr2/u2* describe the string with *char* as the first character. Note that when a space char is given, a tab is also treated as a space.

`: -TRAILING     \ c-addr u1 -- c-addr u2                    17.6.1.0170`

Modify a string address/length pair to ignore any trailing space or tab characters.

`: -leading      \ caddr len -- caddr' len'`

Modify a string address/length pair to ignore any leading space or tab characters.

`: -white        \ caddr len -- caddr' len'`

Remove leading and trailing white space from a string.

`: UPC           \ char -- char'`

Convert supplied character to upper case if it was alphabetic otherwise return the unmodified character. `UPC` is English language specific.

`: UPPER         \ addr len --`

Convert the ASCII string described to upper-case. This operation happens in place. `UPPER` is English language specific.

`: ucmove        \ addr1 addr2 len --`

Copy *len* bytes/characters of memory forwards from *addr1* to *addr2*, converting to upper case. Note that as VFX Forth characters are 8 bit units, there is an implicit connection between a byte and a character.

`: ucmove>       \ addr1 addr2 len --`

Copy *len* bytes/characters of memory backwards starting at *addr1_len-1* to *addr2+len-1*, converting to upper case. Note that as VFX Forth characters are 8 bit units, there is an implicit connection between a byte and a character.

`: umove         \ addr1 addr2 u --`

An intelligent memory move which avoids memory overlap problems. Characters are converted to upper case during the move. Note that as VFX Forth characters are 8 bit units, there is an implicit connection between a byte and a character.

`: uplace        \ c-addr1 u c-addr2 --`

Copy the string described by *c-addr1/u* to an upper-case counted string at *c-addr2*.

`: s=            \ addr1 addr2 len -- flag`

Compare two same-length strings or memory blocks, returning true if they are identical.

`: str=          \ addr1 len1 addr2 len2 -- flag`

Compare two addr/len memory blocks, returning true if they are identical both in length and contents. The comparison is case sensitive.

`: is=           \ c-addr1 c-addr2 u -- flag`

Compare two same-length strings/memory blocks, returning true if they are identical. The comparison is case insensitive.

`: istr=         \ addr1 len1 addr2 len2 -- flag`

Compare two addr/len memory blocks, returning TRUE if they are identical both in length and contents. The comparison is case insensitive.

```
: compare       \ c-addr1 u1 c-addr2 u2 -- n                    17.6.1.0935
```
Compare two strings. The return result is 0 for a match or can be -ve/+ve indicating string differences. If the two strings are identical, n is zero. If the two strings are identical up to the length of the shorter string, n is minus-one (-1) if u1 is less than u2 and one (1) otherwise. If the two strings are not identical up to the length of the shorter string, n is minus-one (-1) if the first non-matching character in the string specified by c-addr1 u1 has a lesser numeric value than the corresponding character in the string specified by c-addr2 u2 and one (1) otherwise.

```
: icompare      \ c-addr1 u1 c-addr2 u2 -- n
```
A case insensitive version of COMPARE.

```
: SEARCH        \ c-addr1 u1 c-addr2 u2 -- c-addr3 u3 f     17.6.1.2191
```
Search the string *c-addr1/u1* for the string (\i{c-addr2/u2}. If a match is found return *c-addr3/u3*, the address of the start of the match and the number of characters remaining in *c-addr1/u1*, plus flag *f* set to true. If no match was found return *c-addr1/u1* and *f=0*. Case sensitive.

```
: instring      \ pattern lenp source lens -- flag
```
Return true if the source text contains the pattern text. Case-sensitive.

```
: $Null         \ -- caddr 0
```
Return a null string.

```
: extractNum    \ caddr len base -- caddr' len' u
```
Extract a number in the given base from the start of the string, returning the remaining string starting at the first non-numeric character and the converted number.

```
: ExtractText   \ caddr len char -- raddr rlen laddr llen
```
Extract text delimited by *char* from the string *caddr/len*. Text before the leading delimiter is ignored. Return the string remaining and string between the delimiters. For example:

```
  s"     'foo' 1 2 10 " char ' ExtractText
```

will return the strings `" 1 2 10 "` and `"foo"`. If either of the delimiters is not present, the original string is returned as *raddr/rlen* and *laddr/llen* is a null string.

```
: csplit        \ addr len char -- raddr rlen laddr llen
```
Extract a substring at the start of addr/len, returning the string raddr/rlen which includes char (if found) and the string laddr/llen which contains the text to left of char. If the string does not contain the character, raddr is addr+len and rlen=0.

4.15.2 Counted strings

```
create cNull    \ -- addr
```
Return the address of an empty counted string.

```
: PLACE         \ c-addr1 u c-addr2 --
```
Copy the string described by c-addr1 u to a counted string at the memory address described by c-addr2.

```
: count         \ addr1 -- addr2 len                            6.1.0980
```
Given the address of a counted string in memory this word will return the address of the first character and the length in characters of the string.

```
: $move         \ caddr1 caddr2 -- ; move counted string
```
Copy a counted string from *caddr1* to *caddr2*. Overlapped strings are handled properly.

```
: SMOVE         \ caddr1 caddr2 --
```

Copy the counted string at *caddr1* to *caddr2*. Overlapped strings are handled properly.

`: addchar        \ char string --`

Add the character to the end of the counted string.

`: append         \ c-addr u $dest --`

Add the string described by C-ADDR U to the counted string at $DEST. The strings must not overlap.

`: $+             \ $addr1 $addr2 --`

Add the counted string $ADDR1 to the counted buffer at $ADDR2.

`: s+             \ source dest --`

Given the addresses of two counted strings, add the source string to the end of the destination string.

4.15.3 Zero-terminated strings

This section provides a set of simple words for handling zero-terminated strings. Additional words can be found in the tools layer.

`create zNull    \ -- addr`

Return the address of a zero terminated null string.

`: zstrlen        \ addr -- len`

Return the length of a 0 terminated string.

`: zcount         \ zaddr -- zaddr len`

A version of `COUNT` for zero terminated strings, returning the address of the first character and the length.

`: zplace         \ addr len zaddr --`

Copy the string addr/len to zaddr as a 0 terminated string.

`: zmove          \ src dst -- ; shows off the optimiser`

Copy a zero terminated string.

`: zAppend        \ caddr len zdest --`

Add the string defined by *caddr/len* to the end of the zero terminated string at *zdest*.

`: Appendz        \ caddr len zdest --`

Add the string defined by *caddr/len* to the end of the zero terminated string at *zdest*. **OBSOLETE**: use `zAppend` above instead. `Appendz` will be removed in a future release.

`: (z$+)          \ caddr u zdest$ --`

Add the source string caddr/u to the end of the zero terminated destination string. **OBSOLETE**: use `zAppend` above instead.

`: z$+            \ zsrc$ zdest$ -- ; add zsrc$ to end of zdest$`

Add the source string to the end of the destination string. Both strings are zero terminated.

4.15.4 Pattern matching

VFX Forth provides a few words that check if a string matches a template string that can have simple wildcards. If you need something more sophisticated, you are probably best off interfacing to a regex library such as the one at

```
www.pcre.org
```

Our thanks to Graham Smith at Tectime for the code.

Take two strings, a 'source' string and a 'pattern'. The test is to see if the source matches the pattern where the pattern can contain the wildcard characters '?' and '*'. These two characters can be 'escaped' using the character '\'.

The asterisk as a wildcard implies 'any of zero or more characters match'. Thus '*' will match with each of 'a', '12abxyz' and the zero length string ''. An asterisk then matches anything. A pattern of "ab*12" will match any text which starts with 'ab' and ends with '12'.

The question mark indicates any one character. Under DOS/Windows the question mark can also match zero characters but this behaviour seems inconsistant - see below for an example. The code here insists that a question mark matches exactly one of any one character. Thus '?' matches 'a', 'b' and '%'. It does not match the zero-length string ''.

```
Source            Pattern          Match
--------------    --------------   -----
"abc"             "abc"            yes
"abcd"            "abc"            no
"abc"             "abc*"           yes
"abc"             "abc?"           yes
"abc"             "*abc"           yes
"abc"             "?abc"           no
"ab"              "a?b"            no
"a"               "?"              yes
"123abc"          "*abc"           yes
"123abc"          "?abc"           no
"123abc"          "???abc"         yes
"123abc"          "1*c"            yes
""                ""               yes
```

`: wcMatch?       \ src slen ptn plen -- t/f`

Wild Card Match. *src* is the address of the start of a souce string and *slen* is its length. Similarly, *ptn* is the address of a pattern string and its length is *plen*. A value of TRUE is returned only if the source string matches the pattern according to the rules described above. The comparison is case sensitive.

`: iwcMatch?      \ src slen ptn plen -- t/f`

As `wcMatch?` above, but the comparison is case insensitive.

`: strRmatch      \ *s lastS il *p lastP jl -- flag`

Return true if the string described by *addr last first* matches the pattern described by a similar set of three parameters. In the set of three parameters (triple), *addr* is the start of the string, *last* is the zero-based index of the last character in the string. and *first* is the zero-based index of the first character. Originally coded as a primitive of `$cstrmatch` and `$strmatch`, this word now converts the two triples to the more standard Forth *addr length* doubles and calls `wcMatch?`.

`: $cstrmatch     \ src srclen patt pattlen -- flag`

A synonym for `WCMatch?`.

`: $strmatch      \ src patt -- flag`

Perform `wcMatch?` on two counted strings.

`: zstrmatch     \ src patt -- flag`
Perform `wcMatch?` on two zero-terminated strings.

DOS/Windows inconsistency

When using the wildcard character '?' in a file path/name matching routine in Windows or DOS, e.g. the DIR shell command, the question mark sometimes matches zero characters. For instance a pattern of 'ab?.*' matches the file nme 'ab.txt'. However, placing the question mark in another position causes the match to fail. For example, the pattern '?ab.*' does not match 'ab.txt'.

4.15.5 SYSPAD buffering

The `SYSPAD` mechanism replaces the use of `PAD` in the kernel. `SYSPAD` is built in the user area of each task and forms a circular buffer of strings. The lifetime of each string is not defined. It will last until another buffer request causes the memory to be reused.

`: getSyspad     \ u -- addr`
Reserve u bytes in the `SYSPAD` area and return the base address.

`: >Syspad       \ caddr len -- caddr' len`
Copy a string to `SYSPAD` and return the new string.

`: >SyspadC      \ caddr len -- caddr'`
Copy a string to `SYSPAD` and return the new counted string.

`: >SyspadZ      \ caddr len -- zaddr`
Copy a string to `SYSPAD` and return the new zero terminated string.

4.16 Formatted and Unformatted number conversion

4.16.1 Tools

`: BELL          \ --`
`EMIT` the ASCII '7' bell character. Not all output devices support this function. The `USER` variable `OUT` is not incremented by this word.

`: SPACE         \ --                                    6.1.2220`
Output a space (ASCII #32) character to the terminal.

`: SPACES        \ n --                                  6.1.2230`
Output 'n' spaces to the terminal, where n>0. For n<=0 no action is taken.

`: >pos          \ +n --`
Place cursor on current line to column n if possible.

`: BS            \ --`
Output a destructive backspace sequence to the terminal. If the cursor is not at column 0, ASCII characters 8, 32 and 8 are `EMIT`ted and the `USER` variable `OUT` is decremented by one.

`: HEX           \ --                                    6.2.1660`
Change current number conversion base to base 16.

`: DECIMAL       \ --                                    6.1.1170`
Change current number conversion base to base 10.

`: BINARY        \ --`

Change current number conversion base to base 2.

4.16.2 Numeric output

These words are used for displaying numbers.

`: HOLD \ char --` 6.1.1670
Insert the ASCII 'char' value into the pictured numeric output string currently being assembled.

`: HOLDS \ caddr len --`
Insert the string *caddr/len* into the pictured numeric output string currently being assembled.

`: SIGN \ n --` 6.1.2210
Insert the ASCII 'minus' symbol into the numeric output string if 'n' is negative.

`: # \ ud1 -- ud2` 6.1.0030
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. N.B. the output string is built from right (lsd) to left (msd).

`: #S \ ud1 -- ud2` 6.1.0050
Keep performing `#` until all digits are generated.

`: <# \ --` 6.1.0490
Begin definition of a new numeric output string buffer.

`: #> \ xd -- c-addr u` 6.1.0040
Terminate defnition of a numeric output string. Returns address and length of the ASCII result.

`: .BYTE \ b --`
Display the byte *b* as a 2 digit hex number.

`: .WORD \ w --`
Display the 16 bit word 'w' as a 4 digit hex number.

`: .LWORD \ dw --`
Display the 32 bit long word 'dw' as an 8 digit hex number. The two groups of four digits are separated by a ':'.

`: .DWORD \ dw --`
An 'Intel-ised' alias for `.LWORD`.

`: .ASCII \ char --`
Output the supplied ASCII character 'char' via `EMIT` if it is a displayable character. Otherwise a period '.' is output.

`: (u.) \ u -- caddr len`
Return the ASCII string corresponding to the unsigned number *u*.

`: (.) \ n -- caddr len`
Create an ASCII string for the the signed number *n*.

`: (u.r) \ u +n -- caddr len`
Return the string corresponding to the unsigned number *u*. The string is right aligned in a field +*n* characters wide.

`: UD.R \ ud n --`
Output the unsigned double number 'ud' using the current `BASE`, right justified to 'n' characters. Padding is inserted using spaces on the left side.

```
: D.R              \ d n --                                       8.6.1.1070
```

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

```
: D.               \ d --                                         8.6.1.1060
```

Output the double number 'd' without padding.

```
: .                \ n --                                           6.1.0180
```

Output the cell signed value 'n' without justification.

```
: U.               \ u --                                           6.1.2320
```

As with `.` but treat as unsigned.

```
: U.R              \ u n --                                         6.2.2330
```

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

```
: .R               \ n1 n2 --                                       6.2.0210
```

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

4.16.3 Numeric input conversion

VFX Forth provides a flexible number conversion system. It is designed for application use as well as for compiling Forth source code.

The ANS and Forth200x Forth standards specify that floating point numbers must be entered in the form 1.234e5 and must contain a point '.' and 'e' or 'E'. Double numbers (integers) are terminated by a point '.'.

This situation prevents the use of the standard conversion words in international applications because of the interchangable use of the '.' and ',' characters in numbers. To ease this, VFX Forth uses two system variables, `FP-CHAR` and `DP-CHAR`, to hold the characters used as the floating point and double number integer indicator characters. By default, `FP-CHAR` is initialised to '.' and `DP-CHAR` is initialised to ',' and '.'. For ANS and Forth200x compliance, you should set them as follows:

```
\ ANS standard setting
  char . dp-char !
  char . fp-char !
: ans-floats    \ -- ; for strict ANS compliance
  [char] . dp-char !
  [char] . fp-char !
;
\ MPE defaults
  char , dp-char c!
  char . dp-char 1+ c!
  0 dp-char 2+ c!
  char . fp-char !
: mpe-floats    \ -- ; for VFX Forth v4.4 onwards
  [char] , dp-char c!
  [char] . dp-char 1+ c!
  0 dp-char 2+ c!
  [char] . fp-char !
;
: mpe-floats    \ -- ; for VFX Forth before v4.4
  [char] , dp-char !
  [char] . fp-char !
;
```

You can of course set these variables to any value that suits your application's language and locale. Note that integer conversion is always attempted before floating point conversion. This means that if `FP-CHAR` and `DP-CHAR` contain the same characters, floating point numbers must contain 'e' or 'E'. If they are different, a number containg a character in `FP-CHAR` will be successfully converted as a floating point number, even if it does not contain 'e' or 'E'.

`: DIGIT          \ char base -- 0 | n true`
If the ASCII value *char* can be treated as a digit for a number within the number conversion base *base*, i.e. in the range 0..base-1, then return the digit and a TRUE/-1 flag, otherwise return FALSE/0.

`: SKIP-SIGN      \ addr1 len1 -- addr2 len2 t/f`
Given the address and length of a string skip a leading plus or minus symbol and return modified address and length. The flag t/f is TRUE if a leading minus was found. From build 2514 onwards, conversion is case insensitive.

`: +DIGIT         \ d1 n -- d2`
Accumulate digit value n into double d1 to form d2 such that d2=d1*base+n.

`: isSep?         \ char addr -- flag`
Return true if *char* is one of the four bytes at *addr*. If less than than four bytes are needed, a zero byte acts as a terminator.

`: +CHAR          \ char -- flag`
The character char is not a digit, so check to see if it is another permitted character in a number such as a double number separator. Return true if char is valid.

`: +ASCII-DIGIT   \ d1 char -- d2 flag`
Accumulate the double number d1 with the conversion of char, returning true if the character is a valid digit or part of an integer.

`: OverrideBase   \ caddr u -- caddr' u'`
Used by `integer?` to force a `BASE` override. See `integer?` below for details.

: <integer?>   \ caddr len -- d 2 | n 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 integer return result above that cell or 2 above a double cell integer. The ASCII number string supplied can also contain number conversion base overrides. A leading $ enforces hexadecimal, a leading # enforces decimal and a leading % a leading '0x' or trailing 'h'. Character literals can be obtained with 'x' where x is the character. A double number contains one of the characters in the variable DP-CHAR, by default ',' and '.'.

: integer?   \ addr -- d 2 | n 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 integer return result above that cell or 2 above a double cell integer. The ASCII number string supplied can also contain number conversion base overrides. A leading $ enforces hexadecimal, a leading # enforces decimal and a leading % enforces binary. Hexadecimal numbers can also be specified by a leading '0x' or trailing 'h'. Character literals can be obtained with 'x' where x is the character. A double number contains one of the characters in the variable DP-CHAR, by default ',' and '.'.

: >NUMBER   \ ud1 c-addr1 u1 -- ud2 c-addr2 u2   6.1.0570

Accumulate digits from string c-addr1/u1 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. From build 1656 onwards >NUMBER is case insensitive.

4.17 More string words

: $.   \ c-addr --

Output a counted string to the output device.

: (")   \ -- a-addr

Return the address of a counted string that is inline after the calling word, and adjust the calling word's return address to step over the inline string. See the definition of (.") for an example. This word is now obsolete and will be removed in a future release.

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

Output the text up to the closing double-quotes character.

variable ^null   \ -- *null

Return a "pointer-to-null" address.

: wcount   \ addr1 -- addr2 len

Given the address of a 16-bit word-counted string in memory WCOUNT will return the address of the first character and the length in characters of the string.

: (W")   \ -- waddr u ; step over caller's in line string

Returns the address and length of inline 16-bit word-counted and 16-bit zero-terminated string. Steps over the inline text to a cell-aligned boundary.

: ((W"))   \ -- waddr u ; dangerous factor!

A factor provided for the generation of long string actions that have to step over an inline string. For example, to define W." which uses a long string, you might compile (W.") and then use W", to compile the inline string. The definition of (W.") then might be:

```
: (W.")     \ --
  ((W")) type
;
```

4.18 Linked lists

`: link,          \ var-addr -- ; lay a link in a chain whose head is at var-addr`
Add a link to a chain anchored at address var-addr. The old contents of var-addr are added to the dictionary as the new link, and the address of the new link is placed at var-addr.

`: AddLink        \ item anchor -- ; add a new item to end of chain, link is first`
Used instead of `LINK,` when a new item in the chain already exists, e.g. it has been `ALLOCATE`d. The item is added to the start of the chain. Note that this word requires the link to be at offset 0 in the item being added.

`: AddEndLink     \ item anchor --`
Add an *item* (a structure) to the end of of the chain anchored at *anchor*. The link field must be at offset 0 in *item*.

`: DelLink        \ item anchor -- ; remove item from chain`
Delete/Remove an item from a chain achored at address anchor. Note that this word requires the link to be at offset 0 in the item being removed.

`: ExecChain      \ anchor --`
Execute the contents of chain with the following structure:

```
link | xt | ...
```

Each word that is run has the stack effect

```
^link -- ^link
```

Where ^*link* is the address of the link field in the structure. Thus, data that follows the xt can easily be accessed.

`: AtExecChain    \ xt anchor --`
Add the word whose *xt* is given to the chain anchored at address *anchor*.

4.19 Wordlists and Vocabularies

Wordlists and vocabularies are described in a separate chapter.

4.20 Input Specification and Parsing

The Forth interpreter operates on a "terminal input buffer". This buffer is parsed space-delimited token by token by the system. Standard words exist for managing the source of the text.

`0 value SOURCE-ID                              \                 6.2.2218`
`SOURCE-ID` describes the method used to refill the terminal input buffer. If the value is "0" the input source is the console, a value of "-1" indicates the input source is a string - via `EVALUATE` - any other value is taken to be a file-id for source inclusion from a text file.

`: TIB            \ -- c-addr                            6.2.2290`
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.

`tib-len constant tib-len        \ -- u`
Returns the size of the console input buffer.

`: SOURCE         \ -- c-addr u                          6.1.2216`

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

: TO-SOURCE \ c-addr u --

Set the address and length of the system terminal input buffer.

: SAVE-INPUT \ -- xn..x1 n 6.2.2182

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.

: RESTORE-INPUT \ xn..x1 n -- flag 6.2.2148

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 \ -- 6.2.2040

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

: REFILL \ -- flag 6.2.2125

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.

: PARSE \ char"ccc<char>" -- c-addr u 6.2.2008

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

An alternative to WORD below. The returned string is a c-addr/u pair rather than a counted string and no copy has occured, i.e. the contents of HERE are unaffected. The returned string is in the input buffer, which should not be modified. Because no intermediate global buffers are used PARSE-WORD is more reliable than WORD for text scanning in multi-threaded applications and in winprocs.

: parse-name \ -- c-addr u ; Forth200x

Equivalent to BL PARSE-WORD above. **Do not** modify the returned string if you want to be compliant with the ANS or Forth200x standards. PARSE-NAME can replace BL WORD COUNT in most cases. Because no intermediate global buffers are used PARSE-NAME is faster and more reliable than WORD for text scanning in multi-threaded applications and in callbacks.

: WORD \ char"<chars>ccc<char>" -- c-addr 6.1.2450

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

: parse-leading \ char --

skip over leading characters of char in the input stream. Tab characters are treated as spaces.

: GET-TOKEN \ "<name>" -- addr

A version of BL WORD in which the returned string is converted to upper case.

: get-word \ char -- c-addr

A version of WORD that works across multiple lines. If a word cannot be obtained, the input stream is REFILLed.

: GetPathSpec \ -- c-addr u | c-addr 0 ; 0 if null string

Parse the input stream for a file/path name and return the address and length. If the name starts with a '"' character the returned string contains the characters between the first and second '"' characters but does not include the '"' characters themselves. If you need to include names that include '"' characters, delimit the string with '(' and ')'. In all other cases a space is used as the delimiting character. `GetPathSpec` does not expand text macro names.

4.21 Runtime and Compile time support for defining words

`defer DOCOLON,          \ --`
Compile the code required at entry to a colon definitions.

`defer DOSEMICOLON,      \ --`
Compile the code required at exit from a colon definitions by `;`.

`defer Compile,          \ xt --                        6.2.0945`
Compile the word specified by xt into the current definition.

`: (;CODE)        \ -- ; R: a-addr --`
Part of the run time action of `;CODE` and `DOES>`, executed when the defining word executes to create a new child word. Patch the last word defined (by `CREATE`) to have the run time action that follows immediately after `(;CODE)`.

`: DOCREATE,      \ --`
Compile the run time action of `CREATE`.

`: LIT            \ -- x`
Code which when CALLED at runtime will return an inline cell value.

`#16 value /code-alignment       \ -- n`
The default code alignment used by `FASTER` below. Must be a power of two.

`#16 value /data-alignment       \ -- n`
The default data alignment used by `FASTER` below. Must be a power of two.

`/code-alignment value code-alignment    \ -- n`
The start of a colon or `CODE` definition is aligned to an alignment boundary defined by this value, which **must** be a power of two.

`/data-alignment value data-alignment    \ -- n`
The start of the data areas defined by CREATE and friends is aligned to a boundary defined by this value, which **must** be a power of two.

`: smaller        \ --`
Selects smaller code using the minimum of alignment.

`: faster         \ --`
Selects faster code using the preset alignment in `/CODE-ALIGNMENT`, which will usually increase speed and the size of the dictionary headers.

`: CODE-ALIGN    \ --`
`ALIGN` filling with breakpoints (used for code boundaries).

`: data-align    \ --`
`ALIGN` filling with breakpoints (used for data boundaries). The alignment is followed by the run-time code for `CREATE` and the data area is then aligned on the boundary.

`: set-compiler  \ xt --`
Set *xt* as the compiler of the `LATEST` definition. `SET-COMPILER` takes the *xt* of the word it is to compile so that information can be extracted from the word. If you use this in a defining word use `INTERP>` rather `DOES>`. See the VFX code generator section of the manual for more details.

: get-compiler \ -- xt
Get *xt* of the compiler of the LATEST definition. If the return value is zero, the word has no compiler.

4.22 Defining words

: (:) \ C: caddr len -- colon-sys ; Exec: i*x -- j*x ; R: -- nest-sys
Begin a new colon definition with the name given by *caddr/len*.

: : \ C: "<spaces>name" -- colon-sys ; Exec: i*x -- j*x ; R: -- nest-sys 6.1.045
Begin a new definition called name.

: :NONAME \ C: -- colon-sys ; Exec: i*x -- j*x ; R: -- nest-sys 6.2.0455
Begin a new colon definition which does not have a name. After the definition is complete the semi-colon operator returns the XT of the newly compiled code on the stack.

: ; \ C: colon-sys -- ; Run: -- ; R: nest-sys -- 6.1.0460
Complete the definition of a new 'colon' or :NONAME word.

: DOES> \ C: colon-sys1 -- colon-sys2 ; R: nest-sys -- 6.1.1250
Begin definition of the runtime-action of a child of a defining word. You may not use RECURSE after DOES>.

: INTERP> \ C: colon-sys1 -- colon-sys2 ; R: nest-sys --
Begin definition of the runtime-action of a child of a defining word that sets a compiler with SET_COMPILER for its children between CREATE and INTERP>. You may not use RECURSE after INTERP>. INTERP> and setCompiler are used to avoid defining words with state-smart run-time actions.

: COMP: \ --
Start a :NONAME word that is the compiler for the previous word.

: >DOES \ xt -- addr
Given the xt of the child of a defining word, return the address of the run-time code.

: Synonym \ "<new-name>" "<curdef>" --
Create a new definition which redirects to an existing one. Normal dictionary searches for <*new-name*> will return the xt of <*curdef*>.

: Alias: \ <"new-name"> <"curdef"> --
A synonym for SYNONYM.

: CONSTANT \ x "<spaces>name" -- ; Exec: -- x 6.1.0950
Create a new CONSTANT called name which has the value x. When NAME is executed the value is returned.

: 2constant \ n1 n2 -- ; Exec -- n1 n2 8.6.1.0830
A double number equivalent of CONSTANT.

: VARIABLE \ "<spaces>name" -- ; Exec: -- a-addr 6.1.2410
Create a new variable called name. When Name is executed the address of the data-cell is returned for use with @ and ! operators.

: 2VARIABLE \ "<spaces>name" -- ; Exec: -- a-addr 8.6.1.0440
A double number equivalent of VARIABLE.

: user \ u "<name>" -- ; Exec: -- addr
Create a new USER variable called name. The 'u' parameter specifies the index into the user-area table at which to place the A -405 THROW occurs if there is no more user space. The VFX kernel

supports 4K bytes of `USER` area space starting at offset 4096. `USER` variables are located in a separate area of memory for each task or callback procedure. They are equivalent to "thread local storage" in Windows parlance. Use in the form:

```
$1000 USER TaskData
```

```
: +USER          \ n "<spaces>name" -- ; Exec: -- user-a-addr
```

Create a new `USER` variable called `name` and reserve N bytes of `USER` space, e.g. `8 CELLS +USER TaskStruct`. N is rounded up to the next CELL boundary. See `USER` above. The use of `+USER` avoids having to keep track of assigned `USER` variable offsets.) `+USER` is non-ANS but for portability is trivially defined by:

```
VARIABLE NEXTUSER
: +USER         \ n -- ; -- addr
  NextUser @ user  aligned NextUser +!
;
```

```
: u#            \ "<name>"-- u
```

Return the index of the `USER` variable whose name follows, e.g.

```
u# S0
```

```
: Buffer:       \ n "name" -- ; [child] -- addr
```

Create a memory buffer called `name` which is 'n' bytes long. When `name` is executed the address of the buffer is returned.

```
: value         \ n -- ; ??? -- ??? ; 6.2.2405
```

Create a variable with an initial value. When the `VALUE`'s name is referenced, the value is returned. Precede the name with `TO` or `->` to store to it. Precede the name with `ADDR` to get the address of the data. The full list of operators is displayed by `.OPERATORS ( -- )`.

```
5 VALUE FOO                 \ initial value of FOO is 5
  FOO .                     \ will give 5
  6 TO FOO                  \ new value is 6
  FOO .                     \ will give 6
  ADDR FOO @ .              \ will give 6
```

```
: 2value        \ x1 x2 -- ; ??? -- ??? ; 6.2.2405
```

Create a cell pair with an initial value. When the `2VALUE`'s name is referenced, the value is returned. Precede the name with `TO` or `->` to store to it. Precede the name with `ADDR` to get the address of the data.

```
: operator      \ n --
```

Define an operator with the given number.

```
: Operator:     \ --
```

Define a new operator with automatic numbering.

```
: op#           \ "name" -- n [int] ; "name" -- [comp]
```

Return or compile the operator number

```
: .Operators    \ --
```

List the operators by number and name.

The standard VFX Forth set of operators is as follows. All of them are supported by children of `VALUE`, but not all are supported by other words that use operators.

```
0 operator default      \ fetch
1 operator ->           \ store
1 operator to           \   "
2 operator addr         \ address operator
3 operator inc          \ increment by one
4 operator dec          \ decrement by one
5 operator add          \ add stack item to contents
6 operator zero         \ set to zero
7 operator sub          \ subtract stack item from contents
8 operator sizeof       \ return item size
9 operator set          \ set to -1
```

The following are provided to ease porting from other systems.

```
5 operator +to          \ add stack item to contents
7 operator -to          \ subtract stack item from contents
```

`: DEFER          \ Comp: "<spaces>name" -- ; Run: i*x -- j*x`

Creates a new `DEFER`ed word. A default action, `CRASH`, is automatically assigned. See `CRASH` and the section on vectored execution.

4.23 Interpreter and Compiler

4.23.1 Tools

These words are mostly used for building new interpreting and compiling words

`: !CSP          \ x --`

Mark the position of the compilation stack pointer for later compile time checking.

`: ?CSP          \ --`

Check that the compilation stack pointer is the same as when last marked by `!CSP`.

`: ?EXEC          \ --`

Perform `#-403 THROW` if not in interpretation state.

`: ?COMP          \ --`

Perform `-14 THROW` if not in compilation state.

`: ?STACK          \ --`

Perform `-4 THROW` if the data stack pointer is out of range.

`: ?UNDEF          \ flag --`

Perform `-13 THROW` if flag is false/0, usually because a word is undefined.

`: [          \ --          6.1.2500`

Switch compiler into interpreter state.

`: ]          \ --          6.1.2540`

Switch compiler into compilation state.

4.23.2 Numeric literals

`: LITERAL          \ Comp: x -- ; Run: -- x          6.1.1780`

Compile a literal into the current definition. Usually used in the form

```
[ <expression ] LITERAL
```

inside a colon definition. Note that LITERAL is IMMEDIATE.

```
: DLITERAL      \ Comp: d -- ; Run: -- d
```
A double number version of LITERAL.

```
: 2LITERAL      \ Comp: x1 x2 -- ; Run: -- x1 x2               8.6.1.0390
```
A two cell version of LITERAL.

```
: DoNumber?     \ c-addr -- Nn .. N1 n | 0
```
Wrapper for Number? Used by the system to add the XREF hook for literals. See Number?.

4.23.3 Forth words

```
: '             \ "<spaces>name" -- xt                          6.1.0070
```
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          6.1.2510
```
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.

```
: 'syn          \ "<spaces>name" -- xt
```
Find the xt of the next word in the input stream. Unlike ' above, if the word is a child of SYNONYM, the xt of the SYNONYM is returned, not the xt of the original word.

```
defer Compile,  \ xt --                                         6.2.0945
```
Compile the word specified by *xt* into the current definition.

```
: EXECUTE       \ xt --                                         6.1.1370
```
Execute the word specified by *xt*.

```
: [COMPILE]     \ "<spaces>name" --                             6.2.2530
```
Compile the **compilation** action of the next word in the input stream. [COMPILE] ignores the IMMEDIATE state of the word. Its operation is mostly superceded by POSTPONE. See also [INTERP] below.

```
: [INTERP]      \ "<spaces>name" --
```
Compile the **interpretation** action of the next word in the input stream. [INTERP] is necessary when you want the interpretation behaviour of words such as S" to be compiled. See also [COMPILE] above.

```
: COMPILE       \ "name" --
```
Used in the form COMPILE <name> inside a colon definition, <name> will be compiled when the colon definition executes. COMPILE is mostly superceded by POSTPONE. **OBSOLETE**: This word will be removed in a future release.

```
: POSTPONE      \ Comp: "<spaces>name" --                       6.1.2033
```
Compile a reference to another word. POSTPONE can handle compilation of IMMEDIATE words which would otherwise be executed during compilation.

4.23.4 Strings and characters

```
: $,            \ caddr len --
```
Lay the string into the dictionary at HERE, reserve space for it and ALIGN the dictionary.

```
: ",            \ "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.

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

An alias for ", added because it is in common use.

: S" \ Comp: "ccc<quote>" -- ; Run: -- c-addr u 6.1.2165

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

: C" \ Comp: "ccc<quote>" -- ; Run: -- c-addr 6.2.0855

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

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

An alias for C". This definition is non-ANS and C" should be used instead. OBSOLETE: will be removed in a future release.

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

A Version of C" which returns the address of a zero-terminated string.

create EscapeTable \ -- addr

Table of translations for \a..\z.

: parse\" \ caddr len dest -- caddr' len'

Parses a string up to an unescaped '"', translating '\' escapes to characters much as C does. The returned translated string is a counted string at *dest* The supported escapes (case sensitive) are:

\a	BEL (alert)
\b	BS (backspace)
\e	ESC (escape, ASCII 27)
\f	FF (form feed, ASCII 12)
\l	LF (ASCII 10)
\m	CR/LF pair - for HTML etc.
\n	newline - CRLF for Windows/DOS, LF for Unices
\q	double-quote
\r	CR (ASCII 13)
\t	HT (tab, ASCII 9)
\v	VT
\z	NUL (ASCII 0)
\"	"
\[0-7]+	Octal numerical character value, finishes at the first non-octal character
\x[0-9a-f][0-9a-f]	Two digit hex numerical character value.
\\	backslash itself
\	before any other character represents that character

: readEscaped \ "string" -- caddr

Parses an escaped string from the input stream according to the rules of parse\" above, returning the address of the translated counted string.

: \", \ "string" --

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.

```
: .\"             \ "ccc<quote>"  --
```
As .", but translates escaped characters using parse\" above.

```
: S\"            \ "string" -- caddr u
```
As S", but translates escaped characters using parse\" above.

```
: C\"            \ "string" -- caddr
```
As C", but translates escaped characters using parse\" above

```
: Z\"             \ "string" -- c-addr
```
As Z", but translates escaped characters using parse\" above

```
: z\",           \ "cc<quote>" --
```
Parse text up to the closing quote and compile into the dictionary at HERE as a zero terminated string. The end of the string is **not** aligned.

```
: CHAR           \ "<spaces>name" -- char                      6.1.0895
```
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        6.1.2520
```
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       \ comp: c-addr1 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.

4.23.5 Comments

```
: \              \ "ccc<eol>"  --                              6.2.2535
```
Begin a single-line comment. All text up to the end of the line is ignored.

```
: (              \ "ccc<paren>"  -- ; ( ... )                  6.1.0080
```
Begin an inline comment. All text up to the closing bracket is ignored.

```
: .(             \ "cc<paren>"  -- ; .( ... )                  6.2.0200
```
A documenting comment. Behaves in the same manner as (except that the enclosed text is written to the console.

```
: ParseUntil     \ c-addr u --
```
Parse the input stream for a white-space delimited string, REFILLing as necessary until the string is found or input is exhausted. Mostly used for block comments. The string compare is case insensitive.

```
: ((             \ -- ; (( ... ))
```
Block comment operator. Any source following this is ignored upto and including the terminator, '))', which must be white space separated.

```
: (*             \ -- ; (* ... *)
```
Block comment operator. Any source following this is ignored upto and including the terminator, '*)', which must be white space separated.

```
: #!             \ -- ; #! /bin/bash
```
Begin a single-line comment. All text up to the end of the line is ignored. This form is provided for Unix-based systems whose shells use #! to specify the program to use with the file.

: StopIncluding \ --

Used in a source file to skip the rest of a file, otherwise behaves like \.

: \\ \ --

A synonymm for StopInclude above.

4.23.6 Text interpreter

: Undefined \ c-addr u --

Default action taken by compiler when a parsed token is not recognised as a word or number.

: EVALUATE \ i*x c-addr u -- j*x 6.1.1360

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

: assess \ i*x c-addr u -- j*x

A version of EVALUATE that saves the current state, switches to interpret mode, interprets the string and then restores state.

: init-quit \ --

Perform the set up required before entering the text interpreter.

defer QuitHook \ --

A place holder for user defined clean up actions after a THROW) occurs in *\fo{QUIT.

4.24 DEFERred words and Vectored Execution

A DEFERred word is defined at one point in the source and can have its action ASSIGNed later both during compile time and at execution time. It is similar to a VARIABLE which has @ EXECUTE appended to its execution semantics.

DEFER words are used to

- avoid forward references
- define words whose actions are modified at run time.

: CRASH \ --

The default action of a DEFERed word. CRASH will THROW a code back to the system.

: DEFER \ Comp: "<spaces>name" -- ; Run: i*x -- j*x

Creates a new DEFERed word. A default action, CRASH, is automatically assigned.

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

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 \ xt "<spaces>name" --

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

: action-of \ "<name" -- xt Forth200x

Returns the xt of the current action of the DEFERred word whose name is given. Use in the form ACTION-OF <deferred-word> if you need to save and later restore the action of a word. The xt returned by ACTION-OF can be used by TO-DO.

: IS \ xt "<spaces>name" -- Forth200x

The candidate with the ANS Forth committee for assignment to DEFERred words. In VFX Forth IS is a synonym for TO-DO above.

```
: BEHAVIOR      \ "<spaces>name" -- xt
```

Returns the xt of the current action of the DEFERred word whose name is given. Since BEHAVIOR is just a synonym for ACTION-OF, BEHAVIOR will be removed in a future release.

```
: DEFER@        \ xt1 -- xt2                        Forth200x
```

Given xt1, the xt of a DEFERred word, return xt2, the action of xt1.

```
: DEFER!        \ xt1 xt2 --                        Forth200x
```

Xt1 becomes the action of the DEFERred word defined by xt2.

4.25 Time and Date

```
0 value dow     \ -- dow ; 0=Sunday
```

Returns the local day of the week, starting at 0=Sunday. This value is updated when TIME&DATE below is called.

```
: time&date     \ -- seconds mins hours day month year
```

Return the operating system local time and date, and set DOW as a side effect.

```
0 value SysDow  \ -- dow ; 0=Sunday
```

Returns the system day of the week, starting at 0=Sunday. This value is updated when SYSTIME&DATE below is called.

```
: systime&date     \ -- seconds mins hours day month year
```

Return the operating system local time and date, and set SYSDOW as a side effect.

4.26 Millisecond timing

Most timing in VFX Forth application uses a millisecond timer provided by the host operating system. The words provided are compatible with those used by MPE's embedded systems. The primary word is ticks which returns a time in milliseconds.

```
defer ms        \ n --
```

Wait for *n* milliseconds. Calls the multitasker through PAUSE.

```
defer ticks     \ -- n
```

Return the system timer value in milliseconds. Treat the returned value as a 32 bit unsigned number that wraps on overflow.

```
: later         \ n -- n'
```

Generates a time value for termination in *n* milliseconds time. Because many applications use a timer value of zero to indicate that a timer is not in use, later never returns a value of zero, and always forces the bottom bit of *n'* to be set to 1.

```
: expired       \ n -- flag ; true if timed out
```

Flag is returned true if the time value *n* has timed out. Calls PAUSE.

```
: timedout?     \ n -- flag ; true if timed out
```

Flag is returned true if the time value n has timed out. Does not call PAUSE, so timedout? can be used in callbacks. In particular, TIMEDOUT? should be used rather than EXPIRED inside timer action words to reduce timer jitter.

4.27 Heap - Runtime memory allocation

The heap memory access wordset is compliant with the ANS Standard. The heap is provided and managed by the host operating system and is only limited by the available memory and/or maximum paging file size. See the later paragraphs for implementation-specific details.

`defer allocate  \ size -- a-addr ior`
Allocate SIZE address units of contiguous data space. If successful an aligned pointer and a 0 IOR are returned. On failure the A-ADDR item is invalid and a non-0 IOR is returned. The contents of newly allocated heap memory are undefined.

`defer resize    \ a-addr newlen -- a-addr ior`
Attempt to resize a block of allocated heap memory to *newlen* size in address units. The contents of the memory block are preserved on a successful resize operation but the address of the memory block may change depending on heap load and the type of resizing requested.

`defer free      \ a-addr -- ior`
Attempt to release allocated memory at A-ADDR back to the system. IOR will return as 0 on success or non-zero for failure.

`: ProtAlloc     \ n -- addr`
A protected version of `ALLOCATE` which `THROW`s on failure.

`: ProtFree      \ addr --`
A protected version of `FREE` which does nothing if addr=0, and `THROW`s on failure.
From VFX Forth 4.0 onwards, the heap system has changed. `ALLOCATE`, `FREE` and `RESIZE` are now directly `DEFER`red to use operating system dependent words.

Under Windows the new heap is much faster but is far less tolerant of programming errors. In particular, releasing the same block twice or `FREE`ing memory you did not `ALLOCATE` may/will lead to a crash with the crash screen showing a fault outside VFX Forth. Newly allocated memory is zeroed and executable.

The Linux man page for **malloc()** says:
"*Crashes in malloc(), free() or realloc() are almost always related to heap corruption, such as overflowing an allocated chunk or freeing the same pointer twice.*"

The `SYSTEM` vocabulary contains `INITVFXHEAP ( -- )` and `TERMVFXHEAP ( -- )` which initialise and destroy the heap. They are in the cold and exit chains. Note that if you are generating a DLL or shared library, these words must be explicitly run as the cold and exit chains are not run before `DLLMAIN`.

5 Dictionary Organisation/Manipulation

The heart of any Forth system is the dictionary. There are two types of word which act on the dictionary. The first are those words which act on definition headers, whilst the second set act on dictionary "data-space."

5.1 Definition Header Structure

Definitions created with any standard defining word except `:NONAME` have a header within the dictionary. The header format is:

```
Link | Ctrl | Count | <name> | Term | Line# | Info | XRef | Len/xt | Cgen
--------------------------------------------------------------------------
Cell | Byte |  Byte | n Bytes| Byte |  Word | Word | Cell |  Cell  | Cell
--------------------------------------------------------------------------
```

`Link` Also called LFA. This field contains the address of the "Ctrl Byte" of the previous word in the same wordlist.

`Ctrl` The Control Byte: The top three bits are all set. The lower five bits are:

```
Bit4    SYNONYM bit
Bit3    Smudge bit
Bit2    Immediate bit
Bit1    ** bit
Bit0    *** bit
```

`Count` Also called the Count Byte. This field contains a byte which is the length of the name.

`<name>` A string of ASCII characters which make up the definition name.

`Term` All definition names are terminated with a 0 ASCII byte.

`Line#` This field holds the line number of the first line of source which built the definition. The actual source file reponsible can be found from the SOURCES structure described in the FILE section of this manual.

`Info` MPE/CCS Reserved field.

`XRef` Pointer to XREF Information

`Len/xt` Binary length of the word, or the xt of the code generator for this word.

`Cgen` Holds the address of additional code generator information.

5.2 Header Manipulation Words

These words allow the manipulation and navigation of dictionary headers.

`#23 constant HEADSIZE   \ -- n`
Return the size of a standard dictionary header minus the name text.

`: .NAME         \ nfa --`
Display a definition's name given an NFA.

`: name> \ nfa -- xt`
Move a pointer from an NFA to the XT.

`: name> \ nfa -- xt`
Move a pointer from an NFA to the XT.

`: ctrl>nfa \ ^ctrl -- nfa`
Move a pointer from the control byte to the name field.

`: nfa>ctrl \ nfa -- ^ctrl`
Move a pointer from the NFA to the control byte field.

`: >name \ xt -- nfa|xt`
Move a pointer from an XT back to the NFA or name-pointer. If the original pointer was not an XT or if the definition in question has no name header in the dictionary the returned pointer will be useless. Care should be taken when manipulating or scanning the Forth dictionary in this way. If *xt* is outside the dictionary, a dummy for `"???"` is returned.

`: >BODY \ xt -- a-addr` 6.1.0550
Move a pointer from an xt to the body of a definition. This should only be used with children of `CREATE`. For example, if `FOOBAR` is defined by `CREATE FOOBAR`, then the phrase `' FOOBAR >BODY` would yield the same result as executing `FOOBAR`.

`: BODY> \ a-addr -- xt`
The inverse of `>BODY`. Note that this word is only valid for children of `CREATE` for which the data area follows the code portion. For words created by `VARIABLE`, `VALUE`, and `BUFFER:` amongst others, the result may/will be invalid, especially if the `+IDATA` switch is in use.

`: >line# \ xt -- addr`
Move a pointer from an XT to the Line# field.

`: >info \ xt -- addr`
Move a pointer from an XT to the header INFO field.

`: >xref \ xt -- xref`
Move a pointer from a supplied XT to the XREF field in the header.

`: >code-len \ xt -- addr`
Move a pointer from an XT to the length/xt field.

`: >code-gen \ xt -- addr`
Move a pointer from an XT to the optimiser source field.

`: ZeroOptData \ cl --`
Zero the optimiser fields at *code-len*.

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

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

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

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

`: name? \ addr -- flag`
Check to see if the supplied address is a valid NFA. A valid NFA satisfies the following:

- Previous byte is hex Ex,
- Previous byte is at an aligned address,
- String has a 0 terminator,
- All characters within string are printable ASCII within range 33..255,
- String Length is non-zero.

`: InOvl?       \ addr1 -- addr2|0`

Returns the overlay address (addr2) if the given address (addr1) is within an overlay, otherwise returns 0.

`: InForth?       \ addr -- flag`

Returns true if the given address is within the Forth dictionary or an overlay.

`: IP>NFA         \ addr -- nfa`

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

`: xtoptimised?  \ xt -- flag`

Is the definition with the given XT optimised?

`: patched?       \ xt -- flag`

Is the definition with the given XT patched/plugged?

`: patchxt        \ xtnew xtold -- ud patchflag`

Patch the code for *xtold* to jump to *xtnew*. Return the first 8 bytes of *xtold* as *ud*. *i{Patchflag} is non-zero if *xtold* had already been patched. All optimisation for *xtold* is disabled.

`: unpatch        \ ud patchflag xt --`

Reverse the effect of `PATCHXT`.

5.3 Definition and Data space access.

These words act upon definition information or dictionary data space.

`: LATEST         \ -- c-addr`

Return pointer past the Link of the last definition. Note that this is NOT the name field, but the control field of the dictionary header. Use `CTRL>NFA` to move to the name field.

`: latest-xt      \ -- xt`

Returns the xt of the last definition to have a dictionary header.

`: SMUDGE         \ --`

Toggle the SMUDGE bit of the latest definition.

`: HideName       \ nfa --`

Hide (make un`FIND`able) the word whose NFA is given.

`: RevealName     \ nfa --`

Reveal (make `FIND`able) the word whose NFA is given.

`: (HIDE)         \ xt --`

Hide the definition from XT. OBSOLETE: Removed, see `HideName` above.

`: (REVEAL)       \ xt --`

Reveal the definition from XT. OBSOLETE: Removed, see `RevealName` above.

`: HIDE           \ --`

Make the last word defined invisible to `SEARCH-WORDLIST`, `FIND` and friends.

`: REVEAL \ --`
Make the last word defined visible to `SEARCH-WORDLIST`, `FIND` and friends.

`: >#THREADS \ wid -- a-addr`
Converts a wid wordlist identifier to address of the cell holding the number of threads in the wordlist.

`: >THREADS \ wid -- a-addr`
Converts a wid wordlist identifier to address of the first thread in the wordlist.

`: WID-THREADS \ wid -- addr len`
Given a wid, return the address and length of its table of threads.

`TRUE value warnings? \ -- flag`
Returns true if redefinition warnings are enabled.

`: +warnings \ --`
Enable redefined warnings.

`: -warnings \ --`
Disable redefined warnings.

`0 value LastNameFound \ -- nfa|0`
Set by `SEARCH-WORDLIST` to contain the NFA of the last word found, or zero if no word was found. Use of `LASTNAMEFOUND` avoids having to use `>NAME` later.

`Defer RedefHook \ --`
A hook available for handling redefinitions. The NFA of the previous word is given by `LastNameFound`, and its xt by `Original-Xt`.

`: (RedefHook) \ --`
The default action of `RedefHook`, which is to display the name of the word being redefined.

`: ($CREATE) \ caddr u --`
Create a new definition in the dictionary with the name described by *caddr/u*. The phrase `S" foobar" ($CREATE)` has the same effect as typing `CREATE foobar` at the console.

`: $CREATE \ c-addr --`
As with `($CREATE)` but takes a counted string.

`: CREATE \ -- ; CREATE <name>`
Create a new definition in the dictionary. When the new definition is executed it will return the address of the definition BODY.

`: IMMEDIATE \ --` 6.1.1710
Mark the last defined word as immediate. Immediate words will execute whenever encountered regardless of `STATE`.

`: Immediate? \ xt -- flag`
Return true if the word at *xt* is immediate.

6 Search Order: Wordlists, Vocabularies and Modules

The definitions within the Forth dictionary are divided into groups called WORDLISTs. A wordlist is identified by a unique number called a WID (Wordlist IDentifier), which is returned when a wordlist is created by the word WORDLIST.

At any given time the system has a "search-order", which is an array of WID values representing the wordlists which are searched. This is the CONTEXT array. The system also uses one WID to contain any new definitions. This is called the CURRENT wid.

Vocabularies are named wordlists. When a vocabulary is created by VOCABULARY <name> a word is built which has a new wordlist. When <name> executes the wordlist replaces the first entry in the search order

Modules are special wordlists for hiding implementation details that should not be modified by application programmers.

6.1 Wordlists and Vocabularies

6.1.1 Creation

```
: WORDLIST      \ -- wid                              16.6.1.2460
```
Create a new wordlist and return a unique identifier for it.

```
: VOCABULARY    \ -- ; VOCABULARY <name>
```
Create a VOCABULARY called <name>. When <name> executes, its wordlist replaces the first entry in the search order

```
: voc>wid       \ xt(voc) -- wid
```
Return the WID from a vocabulary with the XT supplied.

6.1.2 Searching

```
1 value CheckSynonym?   \ -- flag
```
If true, words with the synonym bit set in the header will return the original word's xt, otherwise the xt of the child of SYNONYM will be returned. The setting affects SEARCH-WORDLIST and any words that use it, e.g. ', ['] and FIND.

```
: SEARCH-WORDLIST       \ c-addr u wid -- 0 | xt 1 | xt -1    16.6.1.2192
```
Search the given wordlist for a definition. If the definition is not found then 0 is returned, otherwise the XT of the definition is returned along with a non-zero code. A -ve code indicates a "normal" definition and a +ve code indicates an IMMEDIATE word.

```
: v-find        \ caddr voc-xt -- cfa +/-1
```
An equivalent to SEARCH-WORDLIST used by previous MPE Forths. Moved to *LIB\OBSOLETE.FTH*.

```
: Search-Context        \ c-addr len -- 0 | xt 1 | xt -1
```
Perform the SEARCH-WORDLIST operation on all wordlists within the search order.

```
: FIND          \ c-addr -- c-addr 0 | xt 1 | xt -1          6.1.1550
```
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.

```
: FORTH          \ --                                16.6.2.1590
```
Install Forth Wordlist into search-order.

```
: FORTH-WORDLIST        \ -- wid                     16.6.1.1595
```
Return the WID of the `FORTH` wordlist.

```
: ResetMinSearchOrder   \ --
```
Reset the minimum search-order. The minimum search-order reflects a minimal set of WIDs which make up the search order when `ONLY` is executed.

```
: >MIN-ORDER    \ wid --
```
Add a given WID to the minimum search-order.

```
: GET-CURRENT   \ -- wid                             16.6.1.1643
```
Return the WID for the wordlist which holds any definitions made at this point.

```
: SET-CURRENT   \ wid --                             16.6.1.2195
```
Change the wordlist which will hold future definitions.

```
: GET-ORDER     \ -- widn...wid1 n                   16.6.1.1647
```
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   16.6.1.2197
```
Set the new search-order. The top-of-stack is the number of WIDs to place in the search-order. If N is -1 then the minimum search order is inserted.

```
: ONLY          \ --                                 16.6.2.1965
```
Set the minimum search order as the current search-order.

```
: ALSO          \ --                                 16.6.2.0715
```
Duplicate the first WID in the search order.

```
: PREVIOUS      \ --                                 16.6.2.2037
```
Drop the current top of the search order.

```
: DEFINITIONS   \ --                                 16.6.1.1180
```
Set the current top of the search order as the current definitions wordlist.

```
: VOC?          \ wid -- flag
```
Return TRUE if 'wid' is actually a vocabulary

```
: .VOC          \ wid --
```
Display the name of a vocabulary if the WID is a valid wordlist identifier associated with a vocabulary.

```
: ORDER         \ --
```
Display the current search-order. WIDs created with `VOCABULARY` are displayed by name, others are displayed as numeric representations of the WID.

```
: VOCS          \ --
```
Display all vocabularies by name.

```
: WIDS          \ -- ; display wordlists by address or VOC name
```
Display all created wordlists by address or vocabulary name.

```
: -ORDER        \ wid --
```
Remove all instances of the given wordlist from the `CONTEXT` search order.

```
: +ORDER        \ wid --
```

The given wordlist becomes the top of the search order. Duplicate entries are removed.

```
: ?ORDER          \ wid -- flag
```
Return true if the given wordlist is in the search order.

6.1.3 Removing words

```
: trim-dictionary       \ start end --
```
Unlink all definitions in the memory region from *start* to *end*. See `MARKER` for more details of removing words from the dictionary. The dictionary pointer `DP` and `HERE` remain unchanged.

```
: cut-dictionary        \ start --
```
Unlink all definitions in the memory region from *start* to *HERE*. Reset the dictionary pointer to *start*.

```
: prune:          \ --
```
Starts a nameless definition that is added to the prune chain and is later executed by children of `MARKER`. See `MARKER` for more details. Pruning words are passed the start and end addresses of the region being pruned. The stack action of the definition must be:

```
start end -- start end
```

```
: prunes          \ xt -- ; add xt to prune chain
```
Adds the xt to the prune chain that is executed by children of `MARKER`. See `MARKER` for more details. Pruning words are passed the start and end addresses of the region being pruned. The stack action of *xt* must be:

```
start end -- start end
```

```
: remember:       \ --
```
Starts a nameless definition that is added to the remember chain that is executed by `MARKER`. See `MARKER` for more details. The stack action of the new definition must be:

```
--
```

```
: remembers       \ xt -- ; add xt to remember chain
```
Adds *xt* to the remember chain that is executed by `MARKER`. See `MARKER` for more details. The stack action of *xt* must be:

```
--
```

```
: marker          \ "<spaces>name"                        6.2.1850
```
`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 vocabulary and wordlist vocabulary-based chains. If you need to clean up your data structures, you can add code to do this using the words `PRUNE:`, `PRUNES`, `REMEMBER:` and `REMEMBERS`. When `MARKER` runs, the 'remember' chain words are executed to construct preservation data. When the child of `MARKER`, `<name>`, is run, all the words in the 'prune' chain are executed to remove/restore the data to its previous state.

```
: anew            \ " name" --
```
A variant of `MARKER` that executes a previous child of `MARKER` of the same name if it exists, and then creates the marker. This allows you to place `ANEW FOO` at the start of a source file being debugged so that previous versions of the code are always replaced.

```
: Empty           \ --
```
Remove all words added since the system was loaded or `SAVE`d.

```
: forget          \ "<spaces>name"                        15.6.2.1580
```

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

`: $forget        \ $ --`
`FORGET` the word whose name is the given counted string. See `FORGET` and `MARKER`.

6.1.4 Processing words in a wordlist

`: (MAX-DEF)      \ wid-copy -- addr c-addr`
Returns addr and top definition pointer from a copy of a wordlist. The thread is then truncated by one ready for the next call.

`: WalkWordList   \ xt wid --`
Walk through a wordlist calling the definition XT for each word. The definitions are walked in reverse chronological order. The definition at XT will be passed the THREAD# and NFA. This provides a future-proof method of parsing through a wordlist. It will be supported by future versions of the compiler. The XT definition has the stack form:

```
: MyDef    \ thread# nfa -- flag ; Return TRUE to continue
```

`: WalkAllWordLists      \ xt-to-call --`
Call the given XT for each `WORDLIST`. The callback is given the WID and a flag and will return TRUE to continue the walk or false to abandon it. The FLAG supplied will be TRUE if the WID represents a `VOCABULARY` and FALSE if the WID represents a child of `WORDLIST`.

```
: MyDef    \ wid flag -- t/f ; return TRUE to continue
```

`: WalkAllWords   \ xt --`
Walk through all wordlists calling the given XT for each word. The definitions are walked in reverse chronological order of wordlists and then by reverse chronological order within each wordlist. When run, the XT will be passed the THREAD# and NFA. This provides a future-proof method of parsing through all wordlists. The XT definition has the stack form:

```
: MyDef    \ thread# nfa -- flag ; return TRUE to continue
```

`: traverse-wordlist  \ xt wid -- ; Forth2012`
Walk through the wordlist identified by *wid* calling the definition *xt* for each word. The words in the wordlist are walked in reverse chronological order. The word defined by *xt* ia passed an *nt*, which in VFX Forth is an NFA. The XT definition has the stack form:

```
: MyDef    \ nt -- flag ; Return TRUE to continue
```

`: name>string   \ nt/nfa -- caddr len ; Forth2012`
Given an NT/NFA return the name string.

`: name>interp   \ nt/nfa -- xt ; Forth2012`
Convert an NT/NFA to the corresponding xt.

`: name>compile  \ nt/nfa -- xt1 xt2 ; Forth2012`
Given an NT/NFA return *xt1*, the xt of the word, and *xt2*, the word used to compile it. If *xt1* is immediate, *xt2* is of `EXECUTE`, otherwise it is of `COMPILE,`.

`: CheckDict     \ --`

Check the dictionary for corruption and if corrupt perform a `#-418 THROW`.

`: Xt>Wid        \ xt -- wid|0`

Attempts to locate the wordlist which contains the given XT. This definition is designed for interpreter extensions and tools - it is **not** thread safe or re-entrant!

`: MoveNameToWid \ nfa new-wid -- okay?`

Detach the the word whose *nfa* is given from its wordlist and attach it to the wordlist specified by *new-wid*. The word is attached to the new WID at the correct place in a thread to match its original chronological origin. *Okay?* is returned true if the operation was successful.

`: MoveWordToWid \ xt new-wid -- okay?`

Detach the definition for XT from its current wordlist and reattach to the wordlist specified in NEW-WID. The definition is attached to the new WID at the correct place in a thread to match its original chronological origin. Okay? is returned true if the operation was successful. OBSOLETE - use `MoveNameToWid` instead.

6.2 Source Code Modules

Apart from wordlists and vocabularies, VFX Forth provides 'source modules'. A `MODULE` is a section of source code which handles a given task. Rather than having all the factored 'sub-words' built into the public dictionary, a module exists in its own private wordlist and only provides visible access to those words which have been deliberately `EXPORTED` by the author. This method helps to improve the maintainability of large source projects both for single programmers and for group efforts.

When using this system, the implication is that a function exported by the author will be maintained and not change its meaning or implementation in an 'invisible' manner. Unexported words may change at any time.

For example, a module written by one person for use by another may require a sub-word to lay a string in the dictionary. If initially this word takes a counted string and builds a 0 terminated one in the dictionary it is possible that other sources will use this function for their own use. If at a later date the author of the original module needs to store strings in unicode format due to a change in the overall architecture of the module all other unauthorised uses of the sub-word will break through no fault of the original author. By hiding the mechanics of an API in a module this breakage cannot happen.

6.2.1 Module definition

`: Module        \ <"name"> -- old-current`

Begin the definition of a new source module. Modules can be nested and the `EXPORT`s from any module are placed in the current user definitions vocabulary.

`: End-Module    \ old-current --`

Mark the end of the current module under definition.

`: EXPORT        \ old-current -- old-current ; EXPORT <name>`

Export a module definition into the user's definition wordlist. The dictionary header for the word is relinked from the wordlist in which it was defined to the user's definition wordlist.

```
EXPORT <name>
```

`: Set-Init-Module      \ xt --`

St the initialisation action of a module, which can be triggered by `INIT-MODULE <name>`. Must be executed within a module definition, and the xt must have no stack effect (–).

```
  ' <action> SET-INIT-MODULE
```

```
: Set-Term-Module       \ xt --
```

Set the termination action of a module, which can be triggered by `TERM-MODULE <name>`. Must be executed within a module definition, and the xt must have no stack effect (–).

```
  ' <action> SET-TERM-MODULE
```

6.2.2 Module management

```
: INIT-MODULE   \ "<name>" -- ; INIT-MODULE <name>
```

Calls the initialiser of the module whose name follows.

```
: TERM-MODULE   \ "<name>" -- ; TERM-MODULE <name>
```

Calls the terminator of the module whose name follows.

```
: REQUIRES      \ "<name>" -- ; REQUIRES <name>
```

Specifies by name a module which is required in order to compile the current source code. If the required module is not present compilation is `ABORT`ed.

```
 requires MyModule
```

```
: EXPOSE-MODULE         \ -- ; EXPOSE-MODULE <name>
```

This word will add the private word-list of the module `<name>` to the search order. It is a debugging aid and should only be used as such. Using this method to get at a module's internal definitions defeats the purpose of the module mechanism.

```
 expose-module MyModule
```

6.2.3 An Example Module

The code below defines a module with one public word. The module itself doesn't actually do anything of consequence but it does show the definition syntax.

After compilation the only publically available words will be the two exported at the bottom of the module. All other definitions will be hidden and can only be accessed after an `EXPOSE-MODULE` is executed. In this way the actual implementation of the API can be isolated, only the author needs to worry about it.

```
MODULE  counter

variable counter

: incr-counter          \ --
  1 counter +!
;

: get-counter           \ -- val
  counter @
;

: set-counter           \ val --
  counter !
;

: CounterInitialise     \ --
  0 set-counter
;

: Counter@++            \ -- value
  get-counter incr-counter
;

' CounterInitialise SET-INIT-MODULE
' CounterInitialise SET-TERM-MODULE

EXPORT  CounterInitialise     \ Public word to init
EXPORT  Counter@++            \ Fetch value and incr.

END-MODULE
```

`: WIDInfo       \ wid --`

Display loads of information about a given wordlist

7 Generic IO

Generic IO is the name given to VFX Forth's entire input/output architecture. This system allows for a "device-driver" to be written to a standard format such that the drivers are all interchangeable within the Forth System. As noted later you will see that all standard Forth I/O words (such as `EMIT`) are passed through Generic I/O.)

Under VFX each thread has it's own current input and output stream and can be accessed via the standard Forth IO words and a general purpose wordset which acts upon current thread devices. (See Later) In addition Generic IO also supports a wordset which can use a nominated device directly. This second wordset follows the same naming convention as the current-thread wordset.

7.1 Format of a GENIO Driver

An instance of a Generic Driver is described by the following structure, the address of such a structure is called a SID (structure-identifier).

```
CELL     Device Handle (interpretation depends on device),
CELL     Pointer to function Table (see below),
????     Device Private Data.
```

The function table is a list of execution tokens for words to perform various standard actions. Each action word will receive the SID to operate upon at the top of the data stack.

To be a "Generic Device" the vector table must hold valid entries for:

```
Index      Name            Description
  0        OPEN            Open/Initialise a device.
  1        CLOSE           Close a device.
  2        READ            Read to a block of memory.
  3        WRITE           Write a specified block of memory.
  4        KEY             Perform an action equivalent to Forths
                           KEY definition, (i.e. a blocking character
                           read.)
  5        KEY?            Perform an action equivalent to Forths
                           KEY?, (i.e. any-input-pending?)
  6        EKEY            Supports EKEY
  7        EKEY?           Supports EKEY?
  8        ACCEPT          Added to support FORTH definition of the
                           same name. Read a character stream into
                           a memory buffer.
  9        EMIT            Write a single character to a stream.
 10        EMIT?           Check that EMIT can work.
 11        TYPE            As with Forths TYPE. Write a string of
                           characters to a device.
 12        CR              Perform nearest equivalent action of
                           "carriage return" on the device.
 13        LR              As with CR except for "line-feed"
 14        FF              As with CR except for "form-feed"
 15        BS              As with CR except for "backspace"
 16        BELL            Where applicable to the device emit an
                           audible beep.
 17        SETPOS          Set current position. May reflect screen
                           cursor/file position etc.
 18        GETPOS          Read current position.
 19        IOCTL           Perform a special function. Each device
                           may or may not support various IOCTL
                           codes. The currently assigned function
                           codes used by MPE are documented later.
 20        FLUSHOP         Flush any pending output for device.
 21        RFU/READEX      As READ with additional return of count.
                           Not available on all devices.
```

Devices that require additional functions may add these at the end of the table. If additional functions are added the first three must be as below. It is valid for these to perform no action except to return a zero ior.

```
 22        initialise device    addr len sid -- ior
 23        terminate device     sid -- ior
 24        configure device     sid -- ior ; produces a dialog
```

7.2 Current Thread Device Access

The following definitions act upon the nominated input or output stream for the calling thread. Definitions declared with IPFUNC act upon the current input stream and definitions declared with OPFUNC act upon the current output stream.

`struct gen-sid \ -- len`

Define the generic I/O structure known as a SID. This structure does not include any private data.

```
  cell  field   gen-handle
  cell  field   gen-vector
  0     field gen-private
end-struct
```

`OpenFnid    OPFunc open-gen     \ addr len attribs -- handle/sid ior`

Perform the Generic IO "OPEN" action for current output device.

`CloseFnid   OPFunc close-gen    \ -- ior`

Perform the Generic IO "CLOSE" action for current output device.

`ReadFnid    IPFunc read-gen     \ addr len -- ior`

Perform the Generic IO "READ" action for current input device.

`WriteFnid   OPFunc write-gen    \ addr len -- ior`

Perform the Generic IO "WRITE" action for current output device.

`KeyFnid     IPFunc key-gen      \ -- char`

Perform the Generic IO "KEY" action for current input device. This operation is identical to the Forth word KEY.

`Key?Fnid    IPFunc key?-gen     \ -- flag`

Perform the Generic IO "KEY?" action for current input device. This operation is identical to the Forth word KEY?.

`EKeyFnid    IPFunc ekey-gen     \ -- echar`

Perform the Generic IO "EKEY" action for current input device. This operation is identical to the Forth word EKEY.

`EKey?Fnid   IPFunc ekey?-gen    \ -- flag`

Perform the Generic IO "EKEY?" action for current input device. This operation is identical to the Forth word EKEY?.

`AcceptFnid  IPFUnc accept-gen   \ addr len -- len'`

Perform the Generic IO "ACCEPT" action for current input device. This operation is identical to the Forth word ACCEPT.

`EmitFnid    OPFunc emit-gen     \ char --`

Perform the Generic IO "EMIT" action for current output device. This operation is identical to the Forth word EMIT.

`Emit?Fnid   OPFunc emit?-gen    \ -- flag`

Perform the Generic IO "EMIT?" action for current output device. This operation is identical to the Forth word EMIT?.

`TypeFnid    OPFunc type-gen     \ addr len --`

Perform the Generic IO "TYPE" action for current output device. This operation is identical to the Forth word TYPE.

```
CRFnid      OPFunc cr-gen       \ --
```
Perform the Generic IO "CR" action for current output device. This operation is identical to the Forth word `CR`.

```
LFFnid      OPFunc lf-gen       \ --
```
Perform the Generic IO "LF" action for current output device.

```
FFFnid      OPFunc ff-gen       \ --
```
Perform the Generic IO "FF" action for current output device. This operation is identical to the Forth word `PAGE`.

```
BSFnid      OPFunc bs-gen       \ --
```
Perform the Generic IO "BS" action for current output device.

```
BellFnid    OPFunc bell-gen     \ --
```
Perform the Generic IO "BELL" action for current output device.

```
SetposFnid  OPFunc setpos-gen   \ d mode -- ior ; x y mode -- ior
```
Perform the Generic IO "SETPOS" action for current output device.

```
GetposFnid  OPFunc getpos-gen   \ mode -- d ior ; mode -- x y ior
```
Perform the Generic IO "GETPOS" action for current output device.

```
IoctlFnid   OPFunc ioctl-gen    \ addr len fn# -- ior
```
Perform the Generic IO "IOCTL" action for current output device.

```
FlushOPFnid OPFunc FlushOP-gen  \ -- ior
```
Perform the Generic IO "FLUSH" action for current output device.

```
ReadExFnid  IPFunc ReadEx-gen   \ addr len -- #read ior
```
Perform the Generic IO "READEX" action for current input device.

7.3 IO based on a Nominated Device

Generic IO allows you to perform an action on any device without changing the thread's current Input or Output Channel. All the definitions listed as `xxx-GEN` also have an equivalent definition called `xxx-GIO` which has as top of stack an additional parameter which is the *SID* of the nominated device.

```
OpenFnid   GIOFunc open-gio     \ addr len attribs sid -- handle/sid ior
CloseFnid  GIOFunc close-gio    \ sid -- ior
ReadFnid   GIOFunc read-gio     \ addr len sid -- ior
WriteFnid  GIOFunc write-gio    \ addr len sid -- ior
KeyFnid    GIOFunc key-gio      \ sid -- char
Key?Fnid   GIOFunc key?-gio     \ sid -- flag
EKeyFnid   GIOFunc ekey-gio     \ sid -- echar
EKey?Fnid  GIOFunc ekey?-gio    \ sid -- flag
AcceptFnid GIOFunc accept-gio   \ addr len sid -- len'
EmitFnid   GIOFunc emit-gio     \ char sid --
Emit?Fnid  GIOFunc emit?-gio    \ -- flag
TypeFnid   GIOFunc type-gio     \ addr len sid --
CRFnid     GIOFunc cr-gio       \ sid --
LFFnid     GIOFunc lf-gio       \ sid --
FFFnid     GIOFunc ff-gio       \ sid --
BSFnid     GIOFunc bs-gio       \ sid --
BellFnid   GIOFunc bell-gio     \ sid --
SetposFnid GIOFunc setpos-gio   \ d mode sid -- ior ; x y mode sid -- ior
```

```
GetposFnid GIOFunc getpos-gio   \ mode sid -- d ior ; mode -- x y ior
IoctlFnid  GIOFunc ioctl-gio    \ addr len fn# sid -- ior
FlushOPFnid GIOFunc FlushOP-gio \ sid -- ior
ReadExFnid GIOFunc ReadEx-gio   \ addr len sid -- #read ior
RFUFnid    GIOFunc RFU-gio      \ sid --
InitFnid   GIOFunc init-gio     \ addr len sid -- ior
TermFnid   GIOFunc term-gio     \ sid -- ior
ConfigFnid GIOFunc config-gio   \ sid -- ior
```

7.4 Standard Forth words using GenericIO

The following standard Forth definitions are already vectored through their generic IO equivalents. The SID device handle used comes from the `USER` variables `OP-HANDLE` and `IP-HANDLE`.

```
ACCEPT KEY KEY? EKEY EKEY? EMIT EMIT? TYPE CR
```

Also affected are any I/O words within the ANS Core wordset that use these primitives, such as:

```
. .S SEE U. ) $. F. DUMP EXPECT QUERY
```

7.5 Miscellaneous I/O Words

The following IO words are defined along with Generic IO and will use the standard Generic IO vectors.

`: page           \ --`
Performs a FORM-FEED operation. The effect of this differs from device to device. Binary devices will simply output the #12 character, screen devices will clear the screen and printer devices will move on to the next page.

`: cls            \ --`
An alias for `PAGE` which reads more clearly when using a screen based device.

`: at-xy          \ x y --`
The ANS cursor relocation definition. Attempt to move the cursor to relative position X Y. The actual translation of this varies from device to device since it is implemented with the `SETPOS` generic IO vector.

`: SetIO          \ sid --`
Set the given device as the current I/O device.

`: [IO            \ -- ; R: -- ip-handle op-handle`
Used inside a colon definition **only** to preserve the the current I/O devices before switching them temporarily. Usually used in the form:

```
[io  SomeDev SetIO
  ...
io]
```

`: IO]            \ -- ; R: ip-handle op-handle --`

Used inside a colon definition **only** to restore the the current I/O devices after switching them temporarily. See `[IO` for more details.

```
create szSID    \ -- addr ; used as a property string
```

Within a winproc controlling a device, it is often useful to be able to reference the SID of the device. This is best done by using the **SetProp** Windows call to set a property for the Window - see the *STUDIO* directories for examples. MPE code uses the property string `"SID"` for this, and `szSID` is the address of the zero terminated property string.

7.6 Supplied Devices

The following Generic IO device implementations can all be found in the supplied source library folder *LIB\GENIO*.

7.6.1 Memory Buffer Device

This Generic IO Device uses blocks of memory for input and/or output. The source code is in *LIB\GENIO\Buffer.fth*. This is **not** a circular buffer system. After a buffer has been used, the read and write pointers are **not** reset. You must reset the buffer pointers using the IOCTL functions.

As of VFX Forth build 2380, this code has been overhauled. If this causes you problems, the file *BUFFER.old.FTH* contains the previous (but now unsupported) version. The major changes are:

1. More error checking.
2. The `KEY` operation (and hence `ACCEPT` blocks if no data is available.
3. The `KEY?` operation returns the number of unread bytes.
4. The `EMIT?` operation returns the number of unwritten bytes in the buffer..
5. The close operation is protected if the device is already closed.
6. The GENIO `ReadEx` function is implemented.
7. The `Textbuff-sid` GENIO structure has been documented.
8. IOCTL functions have been added. See later for details.

Note that the `ACCEPT` operation does **not** echo. It is designed for extracting lines from saved input.

In order to create a device use the `TEXTBUFF:` definition given later. `TEXTBUFF:` is compatible with ProForth 2.

When opening a memory device the parameters to `OPEN-GEN` have the following meaning:

`ADDR` Address of memory to use as buffer or ignored if dynamic allocation is required.

`LEN` The maximum length of the memory image.

`ATTRIBS` When zero the ADDR parameter is ignored and LEN bytes of memory are allocated from the heap.

```
struct textbuff-sid     \ -- len
```

Defines the length of a SID for a text buffer device.

```
  gen-sid +             \ handle=buffer, reuse field names of GEN-SID
  int tb-len            \ length of buffer
  int tb-attribs        \ attributes
  int tb-wr             \ address to write to, set by SETPOS, WRITE
  int tb-rd             \ address to read from set by SETPOS, READ
end-struct
```

`: ff-tb         \ sid -- ; page/cls on display devices`

This word is run by `PAGE`, `CLS` and `FF-GEN`. It resets (empties) the buffer.

`: setpos-tb     \ x y mode sid -- ior`

This word is run by `SETPOS-GEN`. Mode controls the x and y input values as follows.

`0` x = #bytes written, y is ignored

`-1` x=col, y=line for next character to be written

`-2` x = #bytes read, y is ignored

`-3` x=col, y=line for next character to be read

`: getpos-tb     \ mode sid -- x y ior`

This word is run by `GETPOS-GEN`. Mode controls the x and y return values as follows.

`0` x = #bytes written, y = 0

`-1` x,y for next character to be written

`-2` x = #bytes read, y = 0

`-3` x,y for next character to be read

`-4` x = addr, y = len of unread data

`-5` x = base address, y = size of data area

`: ioctl-tb      \ addr len fn sid -- ior`

This word is run by `IOCTL-GEN` and `IOCTL-GIO`. *Fn* controls the meaning of *addr*, *len* and the *ior* return value as follows:

`0` Get buffer address: *addr*=0, *len*=0, *ior*=addr.

`-1` Set write pointer: *addr*=0, *len*=offset, *ior*=0.

`-2` Get write pointer: *addr*=0, *len*=0, *ior*=offset.

`-3` Set read pointer: *addr*=0, *len*=offset, *ior*=0.

`-4` Get read pointer: *addr*=0, *len*=0, *ior*=offset.

Device Creation

`: initTextBuffSid       \ addr --`

Initialise a previously allocated SID for a text buffer to the default values.

`: textbuff:     \ "name" -- ; Exec: -- sid`

Create a memory buffer called *name*.

`: SizedTextBuff \ size -- sid|0`

Allocates and opens a SID with a buffer of size *size* bytes, and returns the SID on success or 0 on failure.

`: AllocTextBuff \ -- sid|0`

Allocates and opens a SID with a default 16kb text buffer and returns the SID on success or 0 on failure.

```
: FreeTextBuff  \ sid --
```
Closes the SID and frees memory allocated by `AllocTextBuff` or `SizedTextBuff`.

7.6.2 File Device

This Generic IO Device operates on a disk file for input and/or output. The source code can be found in *Lib\Genio\File.fth*. Neither input nor output are buffered, so that this device should not be used when speed is required.

In order to create a device use the `FILEDEV:` definition given later. `FILEDEV:` is compatible with ProForth 2.

When opening a file device the parameters to OPEN-GEN have the following meaning:

```
ADDR            Address of string for filename.
LEN             Length of string at ADDR.
ATTRIBS         Open flags. These match the ANS r/o r/w etc.
```

The ReadEx function is now implemented.

Device Creation

```
struct /FileDev \ -- len
```
Returns the size of the sid structure for a file device.

```
: initFileDev   \ sid --
```
Initialise the sid for a file device. Mostly used when the structure has been allocated from the heap.

```
: filedev:      \ "name" -- ; Exec: -- sid
```
Create a File based Generic IO device in the dictionary.

7.6.3 NULL Device

This Generic IO Device is used as a bit bucket for unwanted output. When used as input KEY? is always false and and read will never return.

In order to create a device use the NULLDEV: definition given later.

When opening a null device the parameters to OPEN-GEN have no meaning.

Device Creation

```
: nulldev:        \ "name" -- ; Exec: -- sid
```
Create a NULL Generic IO device in the dictionary.

7.6.4 Serial Device

This Generic IO Device operates on a serial port for input and/or output.

In order to create a device use the `SERDEV:` definition given later.

When opening a serial device the parameters to `OPEN-GEN` and `open-gio` have the following meaning:

```
ADDR            Address of configuration string.
LEN             Length of string at ADDR.
ATTRIBS         file fam, usually R/W.
```

The configuration string takes the form:

```
/dev/ttyS0 9600 baud  no parity  8 data  1 stop
```

for Serial Port 0 at 9600 baud, 8 data bits, no parity, 1 stop bit. Only the device name is mandatory. Words and IOCTL functions are available to modify the port setting later. Split baud rates are not supported - you will have to set these yourself. If only the device name is given, the line will be set to 115200 baud, N81 in raw mode. Additional configuration comands are documented later in this section, e.g. for setting the DTR and RTS lines. The configuration string is processed with `BASE` set to `DECIMAL`. USB serial devices are discussed at the end of this section.

When using USB serial devices, the name used varies according to your distribution. The most common names appear to be:

```
/dev/ttyUSBx
/dev/ttyACMx
```

Serial primitives

`struct /serial-sid      \ -- len`

Defines the SID of a serial device.

`struct /termios \ -- size`

A structure corresponding to the termios structure used by **tcgetattr** and **tcsetattr**.

```
  4 field termios.c_iflag       \ input mode flags
  4 field termios.c_oflag       \ output mode flags
  4 field termios.c_cflag       \ control mode flags
  4 field termios.c_lflag       \ local mode flags
  1 field termios.c_line        \ line discipline
  NCCS field termios.c_cc       \ control characters
  3 field termios.padding       \ C aligns everything to 32-bits
  4 field termios.c_ispeed      \ input speed
  4 field termios.c_ospeed      \ output speed
end-struct
```

`: setBaud       \ hertz fildes -- ior ; 0=success`

Set the baud rate for an opened file descriptor.

`: setParity     \ char fildes -- ior ; 0=success`

Set the parity for an opened file descriptor. The character must be one of N,E,O.

`: setData       \ u fildes -- ior ; 0=success`

Set the data size for an opened file descriptor. The data size *u* must be one of 5,6,7,8.

`: setStop       \ u fildes -- ior ; 0=success`
Set the number of stop bits for an opened file descriptor. The value of *i{u} must be one of 1 or 2.

`: setDTR        \ flag fildes -- ior ; 0=success`
Set DTR inactive if *flag* is zero, otherwise set it active.

`: setRTS        \ flag fildes -- ior ; 0=success`
Set RTS inactive if *flag* is zero, otherwise set it active.

`: setUnix       \ sid --`
Set the line to have Unix line handling.

`: setDOS        \ sid --`
Set the line to have Windows/DOS line handling.

`' setBaud SerCfg: baud          \ sid ior baud -- sid ior'`
Used in the configuration string to set the baud rate, e.g.

```
9600 baud
```

`' setData SerCfg: data          \ sid ior u -- sid ior'`
Used in the configuration string to set the number of data bits, e.g.

```
8 data
```

`' setParity SerCfg: parity      \ sid ior u -- sid ior'`
Used in the configuration string to set parity, where *u* is one of the characters N, E, or O. Constants are defined, e.g.

```
no parity
even parity
odd parity
```

`' setStop SerCfg: stop          \ sid ior u -- sid ior'`
Used in the configuration string to set the number of stop bits, e.g.

```
1 stop
2 stop
```

`: 8n1           \ sid ior -- sid ior'`
Used in the configuration string to set the most common case, 8 data bits, no parity, 1 stop bit, e.g.

```
8n1
```

`' setDTR SerCfg: DTR            \ sid ior flag -- sid ior'`
Used in the configuration string to set the DTR line, where *flag* is non-zero for active and zero for inactive.

```
1 DTR
```

`' setRTS SerCfg: RTS            \ sid ior flag -- sid ior'`
Used in the configuration string to set the RTS line, where *flag* is non-zero for active and zero for inactive.

```
1 RTS
```

`: Unix          \ sid ior -- sid ior`
Used in the configuration string. Set the serial line to use LF as the line terminator sequence. CR characters will be ignored by `ACCEPT`.

```
: DOS           \ sid ior -- sid ior
```

Used in the configuration string. Set the serial line to use CR/LF as the line terminator sequence. This can also be used for Macs before OS X, but LF characters will be ignored by `ACCEPT`.

```
: open-Ser    \ addr len attribs sid -- sid ior
```

The string *caddr/len* is split into two. The space delimited left hand side is used as the device, e.g. "/dev/ttyS4" which is opened in raw mode. A default set up of 115200 baud, n81 and Unix line handling is applied, and then the right hand side of the string is parsed. Only the words documented as available in the serial configuration string may be used.

```
: ioctl-ser     \ addr len fn sid -- ior
```

The serial `ioctl` functions provide control over the serial line outputs and Unix/DOS mode handling. Where parameters are shown as *??*, their value is ignored.

```
\ ?? ?? #50 sid -- ior ; Unix mode, LF
\ ?? ?? #51 sid -- ior ; DOS/Windows mode, CR/LF
\ ?? ?? #52 sid -- ior ; Mac mode, CR
\ ?? ?? #53 sid -- ior ; native mode, LF for Unices
\ caddr len #55 sid -- ior ; set string for CR.
\ linechar ignchar #56 sid -- ior ; set input chars for ACCEPT
\ ?? flag #60 sid -- ior ; set DTR, nz=active
\ ?? flag #61 sid -- ior ; set RTS, nz=active
```

Device Creation

```
: initSerDev    \ sid --
```

Initialise the sid for a serial device.

```
: serdev:        \ "name" -- ; Exec: -- sid
```

Create a Serial Port based Generic IO device in the dictionary.

```
serdev: <name>
```

USB serial devices

When using USB serial devices, the name used varies according to your distribution. The most common names appear to be:

```
/dev/ttyUSBx
/dev/ttyACMx
```

There are several methods of finding USB serial ports. The simplest seems to be to unplug the device, then reconnect it, then type the following incantation:

```
dmesg | grep tty
```

where you must have root access. On many systems, e.g. Ubuntu

```
sudo dmesg | grep tty
```

is required. The last few lines should then tell you which USB serial port, e.g. `/dev/ttyUSB0` was selected for your device. If the last tells you that the device is now disconnected, it is probably because of the "brltty bug". Unless you need the Braille TTY access, remove the package *brltty*. Repeat:

```
sudo dmesg | grep tty
```

to check that device remains connected. Some forums suggest that you may also need to create the */dev/ttyUSBx* entries. Do this with:

```
sudo mknod /dev/ttyUSB0 c 188 0
sudo mknod /dev/ttyUSB1 c 188 1
sudo mknod /dev/ttyUSB2 c 188 2
```

Linux serial terminal emulators

The most widely used Linux equivalent to Windows' HyperTerm appears to be *minicom*. It isn't pretty, but it works and is easy to use. There are plenty of others, including GUI ones, but *minicom* is the one we come back to as it is available for nearly all distributions.

7.6.5 XTERM Device

The XTERM Generic IO Device controls an xterm or equivalent device. Facilities are provided for cursor positioning, setting the foreground and background colours, line editing and line history. Cursor positioning uses ANSI escape sequences. Any terminal emulator which supports these sequences, e.g. in ANSI or VT100 mode, should work with this code. A good introduction to ANSI escape sequences may be found at *http://en.wikipedia.org/wiki/ANSI_escape_code*.

In order to create a device use the `XTERM:` definition given later. When opening a file device the parameters to `OPEN-GEN` are unused. For compatibility with future versions please set them to -1, e.g.

```
-1 -1 -1 <sid> open-gio
```

The IOCTL function has the following action

You can set the text foreground and background colours:

```
<fcolour> <bcolour> #10 <sid> IOCTL-GIO drop
```

where *colour* is a colour in the XTERM format. If a colour is set to -1 the existing colour is left unchanged. For XTERMs and VT100/220 compatible terminals, the following colours are standard.

```
0 constant Black
1 constant Red
2 constant Green
3 constant Yellow
4 constant Blue
5 constant Magenta
6 constant Cyan
7 constant White
: +bright  \ color -- color'
\ Convert a colour into its bright version.
  #60 +
;
```

Similarly, terminal positioning control uses ANSI (VT100 and VT220) sequences. If you are connecting using Telnet or other remote access techniques (or even a real terminal), set it to ANSI, VT100 or VT220 compatibility mode.

Line editing is performed using the cursor keys, BS (<- or ^H) to delete before the cursor, and the DELETE keys to delete after the cursor. You can also use ^W and ^R for cursor movement. You can recall lines using the up (previous) and down (next) cursor keys. Lines can be edited after recall. You can also use ^E and ^D instead of the up and down keys. Note that Linux implementations are not consistent in the codes returned by keys such as BS and DELETE.

Unlike other devices, an `XTERM` uses three handles that correspond to **stdin**, **stdout** and **stderr**. By default these are handles 0, 1 and 2 respectively. If you wish to use different handles, you are responsible for their management. Use these handles by setting them into the `/xterm-sid` structure below. By default, the open operation uses the preset handles, which are not closed by the close operation.

The source code can be found in the file *Lib/Lin32/Genio/xterm.fth*, which is compiled during the second stage build. If you change this file, perform a second stage build when you wish to commit to using the changed file.

```
struct /xterm-sid       \ -- len
```

Defines the SID of an xterm console device.

```
  gen-sid +                     \ reuse field names of GEN-SID
  int xs.hIn                    \ input handle
  int xs.hOut                   \ output handle
  int xs.hErr                   \ error handle
  int xs.flags                  \ control flags
                                \ bit 0 - QUIT control
                                \ bit 1 - rfu
                                \ bit 2 - 1=maintain history, 0=none
                                \ bit 3 - 1=history in system ini file
  int xs.hiBuff                 \ address of 64k history buffer
  int xs.hiIndex                \ current history line#
  int xs.hiLowIndex             \ lowest history line#
  int xs.hiSection              \ pointer to INI file section name
end-struct
```

```
: initXtermSid  \ addr --
```

Initialise a `/xterm` structure at *addr*.

```
: xterm:        \ -- ; -- sid ; XTERM: <name>
```

Create a new terminal device.

```
xterm: xconsole \ -- addr
```

VFX Forth console.

```
: init-xcon     \ --
```

Set up to use the `xconsole` device. Performed at start up and compilation.

```
: term-xcon     \ --
```

Shut down the `xconsole` device. Performed at shut down.

7.6.6 Sockets

This Generic IO Device operates on a Linux socket for input or output. General socket programming words are made available in the Forth vocabulary.

In order to create a device use the `SOCKDEV:` definition given later.

When opening a socket device the parameters to `OPEN-GEN` have the following meaning:

```
ADDR          Address of configuration data structure
LEN           connection name zstring
ATTRIBS       0=socket, 1=connect, 2=listen.
```

Sockets API

Many of the Linux socket functions are defined. Note that the **accept** function is accessed by `SACCEPT` to avoid a name clash with the ANS word `ACCEPT`.

`AliasedExtern: saccept int OSCALL accept( int, void *, int *);`
Because the sockets **accept** function has a name clash with the Forth word `ACCEPT` it is made available as `SACCEPT`.

Network order (big-endian) operations

TCP/IP protocols usually send data in what is called network order, which just means most-significant byte first. In memory, numbers are thus stored in big-endian form. The following words provide memory operations for this. These functions have to be capable of fetching 32 bit cells from 16 bit aligned addresses, not just from 32 bit aligned addresses.

`: w@(n)          \ addr -- u16`
Network order 16 bit fetch.

`: w!(n)          \ u16 addr --`
Network order 16 bit store.

`: @(n)           \ addr -- u32`
Network order 32 bit fetch.

`: !(n)           \ u32 addr --`
Network order 32 bit store.

`: w,(n)          \ w --`
Network order W,

`: ,(n)           \ x --`
Network order version of , (comma).

General socket functions in Forth

These words are available in the `FORTH` vocabulary for general socket programming.

`max_path buffer: IPname \ -- addr`
Holds the local computer's name as a zero terminated string.

`2 cells buffer: IPaddress       \ -- addr`

Holds the local computer's IP address as a four byte IPv4 number in network order. A value of 0 indicates that the address is unknown.

`: findLinkIP    \ caddr len addr --`

Find the IP address assigned to the given link, e.g. eth0, and place the link address at *addr*.

`#256 buffer: NetIF$     \ -- addr`

Holds the name of the default network device, usually **eth0**. This is a counted string. The default string is "eth0". Note that recent versions of Fedora use "emx" for Ethernet ports.

`: InitLinuxSockets   \ --`

Initialise sockets; called by the cold chain and during compilation.

`: ?sockerr       \ serr -- ior`

If *serr* is -1, the actual errno value is returned, otherwise zero is returned.

`: writesock      \ c-addr u hsock -- len ior`

Write the buffer to a socket, returning the length actually written and 0 on success, otherwise returning SOCKET_ERROR and the Linux error code.

`: readsock       \ c-addr u hsock -- len ior`

Read into a buffer from a socket, returning the length actually read and 0 on success, otherwise returning SOCKET_ERROR and the Linux error code.

`: pollsock       \ hsock -- #bytes|-1`

Poll a socket and return the number of bytes available to be read.

`: sockReadLen    \ caddr len hsock -- ior`

Read *len* bytes of input from a socket to the buffer at *caddr*, returning ior=0 if all bytes have been read. This is a blocking function which will not return until *len* bytes have been read or an error occurs.

`: bindTo         \ hs af port ipaddr -- res`

A non-BSD function that binds a socket to the given set of address family (af, usually AF_INET), port (port) and IP address (ipaddr). The returned result (res) is 0 for success, otherwise -1. See `BIND`.

`: (Connect)      \ caddr u port# socket -- socket ior`

Attempt to connect to a server. The socket has already been created in the appropriate mode. *Caddr/u* describes the server address either as a name or an IPaddress string and *port#* is the requested port. If *u* is zero, *caddr* is treated as a 32 bit number representing an IPv4 address. On success, the socket and zero are returned, otherwise SOCKET_ERROR and the Linux error code are returned.

`: TCPConnect     \ c-addr u port# -- socket ior`

Attempt to create a TCP socket and connect to a server. */i{Caddr/u} describes the server address either as a name or an IPaddress string and *port#* is the requested port. If *u* is zero, *caddr* is treated as a 32 bit number representing an IPv4 address. On success, the socket and zero are returned, otherwise SOCKET_ERROR and the Linux error code are returned.

`: UDPConnect     \ c-addr u port# -- socket ior`

Attempt to create a TCP socket and connect to a server. */i{Caddr/u} describes the server address either as a name or an IPaddress string and *port#* is the requested port. If *u* is zero, *caddr* is treated as a 32 bit number representing an IPv4 address. On success, the socket and zero are returned, otherwise SOCKET_ERROR and the Linux error code are returned.

Socket device

A socket device is created by `SOCKETDEV: <name>`.

```
SocketDev: SDsid  \ -- addr
```

When opening a socket device the parameters to `OPEN-GEN` have the following meaning:

```
  ADDR            Address of an /SDopen data structure.
  LEN             Address of Windows IP address zstring
                  If 0, /SDopen contains the IP adress.
  ATTRIBS         mode: 0=socket, 1=connect,
```

The following constants define the modes used to open socket:

```
  SD_SOCKET SD_CONNECT SD_LISTEN
```

```
struct /SDopen  \ -- len
```

The structure required for opening a socket Generic I/O device. Not all fields are used by all modes. The */fo{/SDopen} structure is defined as follows:

```
  int SDO.af                \ address family, usually AF_INET
  int SDO.type              \ socket type, e.g. SOCK_STREAM
  int SDO.protocol          \ IPPROTO_TCP ...
  sockaddr_in field SDO.sa  \ SOCKADDR_IN structure
end-struct
```

The `SDO.af` field is AF_INET for all TCP/IP operations. The `SDO.TYPE` field is SOCK_STREAM for TCP or SOCK_DGRAM for UDP. The `SDO.protocol` field is IPPROTO_TCP for TCP or IPPROTO_UDP for UDP. The `SDO.sa` field is a SOCKADDR or SOCKADDR_IN structure (same sizes), defined as follows:

```
struct sockaddr_in      \ -- len
  2 field sin_family        \ address family, usually AF_INET
  2 field sin_port          \ port ; in network order
  4 field sin_addr          \ IP address ; in network order
  8 field sin_reserved      \ RFU
end-struct
```

Note that only the first field is stored in native (little-endian for Intel i32) order. The other fields contain data in network (big-endian) form.

To open a socket, fill in a `/SDopen` structure, and call `OPEN-GEN`. The following example connects to a server.

```
SocketDev: SDsid  \ -- addr ; device

create zserver$   \ -- z$addr ; server name
  z", www.mpeforth.com"

create MySDopen   \ -- addr
  AF_INET ,       \ internet family
  SOCK_STREAM ,   \ connection type
  IPPROTO_TCP ,   \ TCP protocol
  AF_INET w,      \ server family, start of SOCKADDR_IN
  #80 w,(n)       \ server port
  #0 ,(n)         \ server IP address if known
  8 allot&erase   \ reserved
...
MySDopen zserver$ SD_connect SDsid open-gio
```

This will return the sid again and a result code (0=success).

The socket can then be used as the current I/O device.

```
: UseSDsid        \ --
  SDsid dup op-handle !  ip-handle !
;
```

`struct /socket-sid      \ -- len`
Defines the SID of a socket device.

`: sd-flush      \ sid -- ior`
Output to the socket is buffered to avoid running out of Linux buffers. Call `FLUSHOP-GEN ( -- ior )` or `KEY?` to transmit the buffered output.

`: sd-close      \ sid -- ior`
The close function flushes pending output, closes the event object if used, performs shutdown with how=1, and closes the socket.

`: sd-type       \ caddr len sid --`
Buffered output.

`: sd-write      \ caddr len sid -- ior`
Buffered output.

`: sd-emit       \ char sid --`
Buffered output.

`: sd-cr         \ sid --`
Buffered output.

`: sd-key?       \ sid -- #bytes|-1`
The `KEY?` primitive for a socket returns the number of bytes available. If an error occurs, -1 is returned and `KEY` returns CR (ASCII code 13) so that `KEY` and `ACCEPT` do not block. Use the `IOCTL` function if you want to test for a specific error return code. Any buffered output is sent first.

`: sd-key        \ sid -- char`
If an error occurs, CR (ASCII code 13) is returned. Any buffered output is sent first.

`: sd-ioctl       ( addr len fn# sid -- ior )`

The IOCTL primitive for a socket is used to get or set socket status. The following functions are supported by `IOCTL-GEN` for sockets.

```
addr 0 #10 sid -- ior
```

Place the number of bytes available to be read by `recv` at addr.

```
0 0 #11 sid -- ior
```

Set the socket to notify when closed. N.B. This is not currently implemented

```
0 0 #12 sid -- ior
```

Ior is returned non-zero if the socket has been closed. Ior is is returned false (zero) if the socket is still open or notification has not been requested. The socket must be open.

```
state FD_xxx #20 sid -- ior
```

Set the created socket to notify on the FD_xxx flags. If state is zero the socket is set/restored to blocking mode otherwise it is set to blocking mode. N.B. This is not currently implemented

```
state FD_xxx #21 sid -- flags
```

Flags contains FD_xxx bits which indicate what events have occurred from the set rquested by the call above. Flags is is returned false (zero) if no events have been reported or notification has not been requested. The socket must be open. N.B. This is not currently implemented

```
0 0 #22 sid -- ior
```

Reset any notifications returned by function 21 above. Ior is zero for success or the Linux error code. N.B. This is not currently implemented

```
0 0 #23 sid -- ior
```

Stop notification. Ior is zero for success or the Linux error code. N.B. This is not currently implemented

```
0 flags #30 sid -- 0  ; set the device flags
0 0     #31 sid -- flags ; get the device flags
```

The device flags control how some operations behave. Flags is a set of bits as follows:

```
Bit 0 - set to stop echoing during ACCEPT.
```

Device Creation

`: InitSD         \ addr --`

Initialise the data required for a socket device at addr.

`: SocketDev:     \ "name" -- ; Exec: -- sid`

Create a new socket device called `name` in the dictionary.

8 Local variable support

For programming a hosted Forth with a GUI interface and for other significant styles of programming, the ANS Forth specification of local variables is inadequate. VFX Forth and other modern Forth systems provide an alternative notation with more functionality and better readability. A subset of this notation became the basis of the Forth200x local variables proposal. The ANS locals mechanism is supported in VFX Forth for backwards compatibility.

8.1 Extended locals notation

The MPE extended local syntax provides a number of significant benefits to the ANS standard.

- Named inputs are in stack comment order rather than reverse to make source more readable.
- The definition line can declare a number of true local variables for tempory data storage.
- Ability to declare local arrays/buffers for structure definitions etc.

In this implementation, locals are allocated as a frame on the return stack. Note that the word's return address is no longer available.

The following example shows a code extract from a WINPROC, there are the traditional 4 inputs, a local array storing a temporary structure and one output.

```
: WndProc    {: hWnd uMsg wParam lParam | clientrect[ RECT ] -- res :}
  uMessage WM_SIZE =
  if
    hWnd clientrect[ GetClientRect drop    \ Get client rect
    hWndChild @                            \ useto resize child
    #0
    #0
    clientrect[ RECT.right @
    clientrect[ RECT.bottom @
    TRUE MoveWindow drop
    0 exit
  then

  ...... Other Messages ....

  hWnd uMessage wParam lParam DefWindowProc      \ Msg default.
;
```

The following syntax for named inputs and local variables is used.

The sequence:

```
{: ni1 ni2 ... | lv1 lv2 ... -- o1 o2 :}
```

defines named inputs, local variables, and outputs. The named inputs are automatically copied from the data stack on entry. Named inputs and local variables can be referenced by name within the word during compilation. The output names are dummies to allow a complete stack comment to be generated.

- The items between `{:` and `|` are named inputs.
- The items between `|` and – are local variables.
- The items between – and `:}` are outputs.

For compatibility with previous implementations, `{` is accepted in place of `{:` and `}` in place of `:}`. The change to `{: ... :}` took place as a result of the Forth200x standard.

Named inputs and locals return their values when referenced, and must be preceded by `->` or `TO` to perform a store, or by `ADDR` to return the address.

Arrays may be defined in the form:

```
arr[ n ]
```

Any name ending in the '[' character will be treated as an array, the expression up to the terminating ']' will be interpreted to provide the size of the array. Arrays only return their base address, all operators are ignored.

In the example below, a and b are named inputs, a+b and a*b are local variables, and arr[is a 10 byte array.

```
: foo    {: a b | a+b a*b arr[ 10 ] -- :}
  a b + -> a+b
  a b * -> a*b
  cr a+b .  a*b .
;
```

Floating point arguments (inputs) and temporaries are declared by placing `F:` before the name, but not for arrays of floats, which should be declared as above. Floating point locals use the CPU's native FP (80x87) stack, and so are most suitable for use with the *%lib%\ndp387.fth* floating point package. Floating point locals are stored in the extended 80 bit (10 byte) format. This is the default for the *%lib%\ndp387.fth* code. The default action of an FP local is to return its value. The following operators can be applied:

- none - return the value,
- `TO` or `->` - store to the local,
- `ADDR` - return the address of the data,
- `ADD` or `+TO` - add to the value,
- `SUB` or `-TO` - subtract from the value.

```
: foo2 {: a f: f1 b f: f2 | f: f3 f: f4 c d e -- :}
  ...
;
```

The arguments *a* and *b* above are integer arguments taken from the Forth data stack. The arguments *f1* and *f2* are FP arguments taken from the floating point unit. Local values *f3* and *f4* are FP locals and the others are integer locals. An example of using FP locals follows:

```
: foo3  {: f: f1 | f: f2 f: f3 -- :}
  0e0 -> f2  10e0 -> f3  ( noop )
  f1 add f2  f1 sub f3  ( noop )
  f2 f.  f3 f.
;
```

: {   \ --

The start of the traditional brace notation { ... }.

: {:   \ --

The Forth200x name to start the extended local variable notation. Use in the form:

{: ni1 ni2 ... | lv1 lv2 ... -- o1 o2 :}

8.2 ANS local definitions

The ANS locals definitions are provided for use with ANS standard compliant code. The ANS locals system offer limited functionality.

: (LOCAL)   \ Comp: c-addr u -- ; Exec: -- x

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. The word (LOCAL) is intended for the construction of user-defined local variable notations. It is only provided for ANS compatibility.

: LOCALS| \ "<name1> ... <namen> |" --

Create named local variables <name1> to <namen>. At run time the stack effect is (xn..x1 --), such that <name1> is initialised to x1 and <namen> is initialised to 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. All locals created by LOCALS| are single-cell integers. In the example below, a and b are named inputs.

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

8.3 Local variable construction tools

variable LVCOUNT   \ -- addr

Holds the offset in the frame for the next local integer variable.

: FRADJUST   \ size -- offset

Adjust the size of the current local values frame. Used by words that create additional local variables outside a LOCALS| ... | or { ... } notation.

9 Working with Files

9.1 Source file names

The following words are useful when writing your own tools.

`: .SourceName   \ ^SFSTRUCT --`

Given a source file structure such as that held by the variable `'SourceFile` display the current file name.

`: CurrSourceName        \ -- c-addr u`

Returns the current source file name **without** expanding any text macros.

`: stripFilename \ cstring --`

The input is a counted string containing a full path and filename e.g. "C:\WINDOWS\SYSTEM32\COMMAND.COM". The file name is removed to leave "C:\WINDOWS\SYSTEM32". Note that the actual directory separator used depends on the host operating system.

9.2 ANS File Access Wordset

The basis for all file operations comes from the ANS specification wordset for Files. The following group of definitions are implementations of the ANS standard set.

The following data types are used:

`fam` "File Access Method", describes read/write permission etc.

`ior` "IO Result", A return result from most IO calls, this value is 0 for success or non-zero as an error-code.

`fileid` "File Identifier", a handle for a file.

9.3 The actual ANS Wordset

`: bin           \ fam -- 'fam`

Modify a file-access method to include BINARY.

`: r/o           \ -- fam`

Get ReadOnly fam

`: w/o           \ -- fam`

Get WriteOnly fam

`: r/w           \ -- fam`

Get ReadWrite fam

`: Create-File   \ c-addr u fam -- fileid ior`

Create a file on disk, returning a 0 ior for success and a file id. Macro names are expanded before the operating system file create call is made.

`: Open-File     \ c-addr u fam -- fileid ior`

Open an existing file on disk. Macro names are expanded before the operating system file open call is made.

`: Close-File    \ fileid -- ior`

Close an open file. Use correct method for VFCACHED files.

`: Write-File    \ caddr u fileid -- ior`
Write a block of memory to a file.

`: write-line    \ c-addr u fileid -- ior`
Write data followed by EOL. IOR=0 for success. Note that the end of line sequence is given by `EOL$` and is operating system dependent.

`: Read-File     \ caddr u fileid -- u2 ior`
Read data from a file, use VF-CACHE Version where appropriate. The number of characters actually read is returned as u2, and ior is returned 0 for a successful read.

`: read-line     \ c-addr u1 fileid -- u2 flag ior       11.6.1.2090`
Read an ASCII line of text from a file into a buffer, without EOL. Read the next line from the file specified by fileid into memory at the address *c-addr*. At most *u1* characters are read. Up to two line-terminating characters may be read into memory at the end of the line, but are not included in the count *u2*. The line buffer provided by *c-addr* should be at least *u1+2* characters long.
If the operation succeeds, *flag* is true and *ior* is zero. If a line terminator was received before *u1* characters were read, then *u2* is the number of characters, not including the line terminator, actually read ($0 <= u2 <= u1$). When $u1 = u2$, the line terminator has yet to be reached.

If the operation is initiated when the value returned by `FILE-POSITION` is equal to the value returned by `FILE-SIZE` for the file identified by *fileid*, *flag* is false, *ior* is zero, and *u2* is zero. If *ior* is non-zero, an exception occurred during the operation and *ior* is the I/O result code.

An ambiguous condition exists if the operation is initiated when the value returned by `FILE-POSITION` is greater than the value returned by `FILE-SIZE` for the file identified by *fileid*, or if the requested operation attempts to read portions of the file not written.

At the conclusion of the operation, `FILE-POSITION` returns the next file position after the last character read.

`: file-size     \ fileid -- ud ior`
Get size in bytes of an open file as a double number, and return ior=0 on success.

`: file-position \ fileid -- ud ior`
Return file position, and return ior=0 on success.

`: Reposition-File       \ ud fileid -- ior`
Set file position, and return ior=0 on success.

`: Resize-File   \ ud fileid -- ior`
Set the size of the file to *ud*, an unsigned double number. After using `RESIZE-FILE`, the result returned by `FILE-POSITION` may be invalid. Note that for a VF-CACHEd file, this operation is performed on the underlying physical file.

`: delete-file   \ caddr len -- ior`
Delete a named file from disk, and return ior=0 on success.

`: FileExist?    \ caddr len -- flag`
Look to see if a specified file exists, returning TRUE if the file exists.

`: file-status   \ caddr len -- x ior                    11.6.2.1524`
Return the status of the file identified by the character string *c-addr/len*. If the file exists, *ior* is zero; otherwise *ior* is the implementation-defined I/O result code. *X* contains implementation-defined information about the file (always zero for VFX Forth).

```
: rename-file   \ caddr1 len1 caddr2 len2 -- ior         11.6.2.2130
```
Rename the file named by the character string *c1addr/len1* to the name in the character string *caddr2/len2*. *Ior* is the I/O result code.

```
: flush-file    \ fileid -- ior
```
Flush changed file data to disk, and return ior=0 on success.

```
: include-file          \ file-id --
```
Include source code from an open file whose file-id (handle) is given. The file is closed by `INCLUDE-FILE`.

```
: included      \ c-addr u --
```
Include source code from a file whose name is given by *c-addr/u*.

```
: include       \ "<name>" --
```
A more convenient form of `INCLUDED`. Use in the form:

```
INCLUDE <name>
```

See `GetPathSpec` for a discussion of file name formats including those containing spaces.

```
: get           \ "<name>" --
```
A synonym for `INCLUDE`. Windows only. OBSOLETE: has been removed. Please replace all uses of `GET` with `INCLUDE` as soon as possible. Note that the name `GET` is used by semaphore code in many libraries.

```
: required      \ c-addr u --
```
If the file specified by *c-addr/u* has already been `INCLUDED`, discard *c-addr/u*; otherwise, perform the function of `INCLUDED`. You must provide the source file's extension.

```
: require       \ "<name>" --
```
Skip leading white space and parse name delimited by a white space character. Put the address and length of the name on the stack and perform the function of `REQUIRED`. You must provide the source file's extension.

9.4 File Caching

VFX Forth supports memory caching of read-only files. Any file which is to be cached is opened using `VF-OPEN-FILE` rather than the ANS word `OPEN-FILE`. The normal ANS wordset can then be used with re-vectoring being automatic. The control directive `+VFCACHE` (see later) enables `INCLUDE` and friends to use file caching automatically, which decreases compilation time for larger projects.

```
: IsFileIDCached?       \ fileid -- flag
```
Determine if an open file referenced by FILEID is a cached file.

```
: VF-Open-File          \ caddr len fam -- fileid ior
```
Open a file using VFCACHE Mode. This means read the whole file into memory.

```
: VF-Close-File         \ fileid -- ior
```
Close a VFCACHED file, i.e. free its memory.

```
: VF-Read-File          \ caddr u fileid -- u2 ior
```
Read into a buffer from a VFCached file.

```
: Mem-Open-File \ c-addr u fam -- fileid ior
```
Open a memory block *caddr/u* using VFCACHE mode. *Fam* is ignored. When this file is closed, no attempt is made to `FREE` *caddr/u*.

```
: IncludeMem     \ c-addr u --
```

Include source code from a memory buffer. Errors cause a `THROW`.

9.5 "Smart File" Inclusion

Any pathname used to include source from a text-file passes through the Smart File filter. This code attempts to resolve the file extension for a name passed to it. The resolve algorithm looks for the file path as specifed, then with a number of common file extensions. See the `ResolveIncludefilename` definition below. If no match is found then the original name is passed back.

```
TRUE value bSmartFileLookUp?
```

When non-zero, the smart file filter is enabled. See also `+SMARTINCLUDE` and `-SMARTINCLUDE` which should be used to control the smart file filter.

```
: dirChar?       \ char -- flag
```

Returns true if the character is one of the two directory separators specified in the system variables `DIR1-CHAR` and `DIR2-CHAR`.

```
: Extension?     \ c-addr u -- len true | false
```

Treats *c-addr/u* as a file name and returns the extension length and true if the file name has an extension (i.e. it ends in '.xxx'), or just false if no extension is present. The extension can be of any length (including 0) as names of the form "name." are treated as having an extension. Unfortunately such names can exist. A name of zero length returns false.

```
: ChangeEXT3     \ c-addr u c-addr1 u1 -- c-addr u
```

Change the last 3 characters of the string at c-addr u to use the text at c-addr1 u1 (where u1 is always 3).

```
: ResolveIncludeFileName        \ c-addr u -- c-addr u
```

Given what may be a extension-less filename attempt to locate a matching file and return its string description. Note that the returned string is built at `HERE`. Matching rules are:

- If the file name exists, return
- If an extension is present, return.
- Look for a recognized extension.

The extensions ".BLD" ".FTH" ".F" ".CTL" ".SEQ" are searched for in that order. For case-sensitive file systems, lower case extensions are tried before upper case. Mixed case is not attempted.

9.6 Source File Tracking

VFX Forth automatically keeps track of compiled source files. Whenever a new source is compiled into the system, the file location and dictionary impact is recorded. One use of this system is `LOCATE` specified below which can attempt to find the source for a definition and automatically load it into your favourite editor for review.

Many users keep their source code in a path (directory or folder) with all the files being loaded by a control file which contains many lines of the form:

```
include part1\petrol
include part1\gas
include part2\forms
include part2\recalculate
...
```

If the source code is moved, for example to a laptop, the new path may be different and LOCATE and friends may then fail. In order to cope with this, additional tracking text can be added at the start of the file name. This text is usually a macro name. What text is added is controlled by the value BuildLevel and the macro DEVPATH.

If BuildLevel is set to 0, no additional information is added. If BuildLevel is set to -1, the contents of the macro DEVPATH are prepended to the file name. **Do not** set BuildLevel to any other values!

DEVPATH may itself contain a macro name. LOCATE expands macros before attempting to open the file. This enables you to partition an application across several build phases, and still be able to LOCATE words when the tree structures have been moved or modified.

`0 value BuildLevel      \ -- n`
Used to control what is added to the start of the file name for source file tracking. See above for more details.

`: +source-files \ --`
Enable source file tracking.

`: -source-files \ --`
Disable source file tracking.

`defer sourceTrackRename \ zaddr --`
A hook so that names for the source file tracking system can be updated to suit user habit. The input *zaddr* is a pointer to a buffer containing a zero-terminated file name. The updated name must be returned in the same buffer. The buffer is of size MAX_PATH bytes. The default action is drop.

`: AddSourceFile \ c-addr u -- 'c-addr 'u ^SFSTRUCT | c-addr u -1`
Add a source file to the tracking vocabulary. *caddr/u* represents the pathname supplied to INCLUDED.

`: (whereis)     \ xt -- c-addr u line# TRUE | FALSE`
Given the XT of a word this will return the filename string, the line number and TRUE for the definition. If the xt cannot be found, just a 0 is returned.

`: whereis       \ -- ; WHEREIS <name>`
Use in the form WHEREIS <name> to find the source location of a word.

`: source-info   \ c-addr u -- start end size true | false`
Return dictionary start/end and binary size of a compiled source file from a string. Returns FALSE only if the source name was not recognized.

`defer .locate   \ --`
Perform the desired action of LOCATE below. The LOCATE_PATH and LOCATE_LINE macros have been set up.

`defer .nolocate \ --`
Perform the action of LOCATE below when the word has been found but has no source information.

```
: LocateInfo    \ caddr u line# --
```
Set the locate macros using *caddr/u* as the file name and *line#* as the line number. The file name is expanded.

```
: locate        \ <"name"> --
```
Use in the form `LOCATE <name>` and display its source code. This word is redefined by the Windows Studio environment.

```
: .sources      \ --
```
Display list of sources used in build so far, includes size, source file name and dictionary pointers.

9.7 Control Directives

The following words can be used to control the filesystem extensions.

```
: +VFCACHE              \ --
```
Enable caching of read-only files when opened.

```
: -VFCACHE              \ --
```
Disable caching of read-only files.

```
: +SMARTINCLUDE         \ --
```
Enable smart resolution of file extensions when including sources.

```
: -SMARTINCLUDE         \ --
```
Disable smart resolution of file extensions when including sources.

```
: +VERBOSEINCLUDE       \ --
```
Enable verbose mode for file includes and overlay handling.

```
: -VERBOSEINCLUDE       \ --
```
Disable verbose mode for file includes and overlay handling.

10 Tools and Utilities

10.1 Conditional Compilation

The following words allow the use of `[IF] ... [ELSE] ... [THEN]` blocks to control which pieces of code are compiled/executed and which are not. These words behave in the same manner as compiled definitions of `IF ... ELSE ... THEN` structures but take immediate effect even outside definitions. Nesting is supported.

`VOCABULARY Compilation? \ --`
The `COMPILATON?` vocabulary holds the control code for passover operations during a conditional block.

`: PassOver      \ n --`
Skip n levels of nested conditional code.

`: have          \ "<name>" -- flag`
Look to see if the word exists in the `CONTEXT` search order and return flag true if found.

`: [defined]     \ "<name>" -- flag` Forth200x
Look to see if the word exists in the `CONTEXT` search order and return flag TRUE if the word exists. This is an immediate version of `HAVE`.

`: [undefined]   \ "<name>" -- flag` Forth200x
The inverse of `[DEFINED]`. Return TRUE if `<name>` does not exist.

`: [ELSE]        \ --` 15.6.2.2531
Marks the start of the ELSE clause of a conditional compilation block.

`: [IF]          \ flag --` 15.6.2.2532
Marks the start of a conditional compilation clause. If flag is TRUE compile/execute the following code, otherwise ignore all up to the next `[ELSE]` or `[THEN]`.

`: [THEN]        \ --` 15.6.2.2533
Marks the end of a conditional compilation clause.

`: [ENDIF]       \ --`
Marks the end of a conditional compilation clause.
The following definitions exist in the `Compilation?` vocabulary.

`: [IF]          1+ ;`
`[IF]` increments the number of levels to skip.

`: [THEN]        1- ;`
`[THEN]` decrements the number of levels to skip.

`: [ENDIF]       1- ;`
`[ENDIF]` decrements the number of levels to skip.

`: [ELSE]        1- dup if 1+ then ;`
`[ELSE]` switches the redirection level.

10.2 Console and development tools

The following words provide useful diagnostic routines and/or general purpose functions in the spirit of the ANS Forth TOOLS and TOOLS EXT wordsets.

`: .tabword      \ addr$ --`

Displays tabbed string, CRing if required. Variable TABWORDSTOP contains the size of a tab.

```
: .tabwordN     \ addr$ --
```

Displays tabbed *NAME*, CRing if required. Variable TABWORDSTOP contains the size of a tab.

```
0 value PauseConsole    \ -- device
```

Some tools, e.g. WORDS and DUMP will pause periodically if PauseConsole returns the same value as the output device in OP-HANDLE. Any interactive console can select this behaviour with:

```
op-handle @ to PauseConsole
```

You can stop any pausing with:

```
0 to PauseConsole
```

```
: flushKeys     \ --
```

Flush any pending input that might be returned by KEY.

```
: HALT?         \ -- flag
```

Used in listed displays. This word will check the keyboard for a pause key (<space> or <lf> or <cr>). If a pause key is pressed it will then wait for another key. The return flag is TRUE if the second key is not a pause key. If the first key is not a pause key TRUE is returned and no key wait occurs. Line Feed characters are ignored.

```
: DUMP          \ addr u --                                 15.6.1.1280
```

Display an arbitrary block of memory in a 'hex-dump' fashion which displays in both HEX and printable ASCII.

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

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

```
: .S            \ --                                        15.6.1.0220
```

Display to the console the current contents of the data stack. If the number base is not HEX than a dump is also made in HEX.

```
: .rs           \ --
```

Display to the console the current contents of the return stack. Where possible a word name is also displayed with the data value.

```
: ?             \ a-addr --                                 15.6.1.0600
```

Display the contents of a memory location. It has the same effect as @ ..

```
: WORDS         \ --
```

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

```
: .FREE         \ --
```

Text display of size of unused dictionary area in Kbytes

```
: mat           \ -- ; MAT <wildcardpattern>
```

Search the current search-order for all definitions whose name matches the wild-carded expression supplied. Expressions can contain either an asterix '*' to match 0 or more characters, or can be a query '?' to mark any single character.

```
: similars      { | temp[ MAX_PATH ] -- }
```

A slightly faster version of MAT with a limited range. The definitions listed will contain <pattern> within their name. <pattern> can only contain printable ASCII characters.

```
: sim           \ -- ; SIM <pattern>
```

A slightly faster version of MAT with a limited range. The definitions listed will contain <pattern> within their name. <pattern> can only contain printable ASCII characters. A synonym for SIMILARS.

10.3 Zero Terminated Strings

A group of simple primitive words to work with 0 terminated ASCII strings.

`: caddr>zaddr    \ caddr zaddr --`
Copy a counted string to a 0 terminated string.

`: .z$            \ zaddr --`
`TYPE` a zero terminated string.

`: .z$EXPANDED            \ zaddr --`
`TYPE` a zero terminated string after macro expansion.

`: z$,            \ c-addr u --`
Lay the given string in the dictionary as a zero terminated string. The end of the string is not aligned.

`: $>z,           \ addr --`
Lay a zero terminated string in the dictionary, given a counted string. The end of the string is not aligned.

`: z",            \ "cc<quote>" --`
"comma" in a zero terminated string from the following text. The end of the string is not aligned.

`: $>ASCIIZ       \ caddr -- zaddr`
Convert a counted string to a zero terminated string. The converted string is in a thread-local buffer.

`: asciiz>$       \ zaddr -- caddr`
Convert a zero terminated string to a counted string. The conversion happens in place.

`: z>here         \ --`
Lay the counted string at `PAD` into the dictionary at `HERE`.

10.4 Structures

The data structure words implement records, fields, field types, subrecords and variant records.

The following syntax is used:

```
STRUCT <name>
  n FIELD <field1>
  m FIELD <field2>
  SUBRECORD <subrec1>
    a FIELD <sf1>
    b FIELD <sf2>
  END-SUBRECORD
END-STRUCT
```

A structure may contain multiple subrecords, and subrecords may be nested.

A field adds its base offset to the given address [that of the record or subrecord]. A record returns its length, and so can be used as an input to field.

```
  len FIELD <name>
n len ARRAY-OF <name>
```

Subrecords are checked for stack depth, like branch structures. They may be nested as required.

Variant records describe an alternative view of the current record or subrecord from the start to the current point. The variant need not be of the same length, but the larger is taken

```
SUBRECORD <name>
   ----
  VARIANT <name2>  ..............  END-VARIANT
END-SUBRECORD
```

use

```
<structure> BUFFER: <name>
```

to create a new instance of a previously defined structure.

The VFX structures package has also been enhanced to handle areas of overlapping data called "UNIONS". Consider the example:

```
   struct test
     int a
     int b
     union
       int c
       int d
      part
       1 field e1
       1 field e2
      part
       int f
       subrecord jim
         float jim1
         int jim2
       end-subrecord
     end-union
     20 field g
   end-struct
```

Each part of a union is overlapped, but fields within a part are treated as individual items. So, in the above example, c and f refer to the same cell, but c and d refer to different cells.

`: struct        \ -- addr 0 ; -- size`
Begin definition of a new structure. Use in the form `STRUCT <name>`. At run time `<name>` returns the size of the structure.

`: end-struct    \ addr n --`
Terminate definition of a structure.

```
: field          \ n <"name"> -- ; Exec: addr -- 'addr
```

Create a new field within a structure definition of size n bytes.

```
: int            \ <"name"> -- ; Exec: addr -- 'addr
```

Create a new field within a structure definition of size one cell.

```
: array-of       \ n #entries size -- n+(#entries*size)
```

Create a new field within a structure definition of size #entries*size.

```
: subrecord      \ n -- n csp 0  [parent]
```

Begin definition of a subrecord.

```
: end-subrecord \ n csp len -- n+len
```

End definition of a subrecord.

```
: variant        \ n -- n csp 0
```

Currently an alias for `subrecord`. Begin a variant.

```
: end-variant    \ n csp m -- n|m
```

Terminate a variant clause.

```
: union          \ currentOffset -- csp 0 currentOffset currentOffset
```

Begin `UNION` definition block.

```
: part           \ max base last -- max base start
```

Begin definition of alternative data description within a `UNION`.

```
: end-union      \ csp maxLength baseOffset lastOffset -- next-offset
```

Mark end of a `UNION` definition block.

```
: field-type     \ n --
```

Define a new field type of size n bytes. Use in the form `<size> FIELD-TYPE <name>`. When `<name>` executes used in the form `<name> <name2>` a field `<name2>` is created of size n bytes.

10.4.1 Forth200x structures

The Forth200x standards effort has adopted s notation that is compatible with VFX Forth, but changes some names.

```
: begin-structure        \ -- addr 0 ; -- size
```

Begin definition of a new structure. Use in the form `BEGIN-STRUCTURE <name>`. At run time `<name>` returns the size of the structure. The Forth200x version of the MPE word `struct`.

```
: end-structure          \ addr n --
```

Terminate definition of a structure. The Forth200x version of the MPE word `end-struct`.

```
: +FIELD         \ n <"name"> -- ; Exec: addr -- 'addr
```

Create a new field of size *n* bytes within a structure definition. The Forth200x version of the MPE word `field`.

```
: cfield:        \ n1 <"name"> -- n2 ; Exec: addr -- 'addr
```

Create a new field of size `1 CHARS` within a structure definition,

```
: field:         \ n1 <"name"> -- n2 ; Exec: addr -- 'addr
```

Create a new field of size `1 CELLS` within a structure definition. The field is `ALIGNED`.

10.5 ENVIRONMENT queries

The ENVIRONMENT system was defined by ANS Forth to enable you to find out about the underlying Forth system. The needs of modern portable libraries have proven the ENVIRONMENT system to be inadequate and so it is little used. The ENVIRONMENT system may be removed in a future standard.

You use the system through the word `ENVIRONMENT?`

```
 caddr len -- false | i*x true
```

where *caddr/len* represents the name of a query. If the system does not know this query, it just returns false (0). If it does know the query, it return the relevant value with true (-1) on top of the stack.

In VFX Forth, `ENVIRONMENT?` is implemented by searching a vocabulary called `ENVIRONMENT`. If the query is found, it is executed.

10.5.1 Predefined queries

The words in this section are defined in the `ENVIRONMENT` vocabulary.

`#255 constant /COUNTED-STRING   \ -- n`
Maximum length of a counted string.

`picnumsize constant /HOLD       \ -- n`
Maximum size of HOLD area.

`padsize constant /PAD           \ -- n`
Maximum size of PAD.

`8 constant ADDRESS-UNIT-BITS    \ -- n`
Number of bits in an address unit (byte in this system).

`true constant CORE              \ -- TRUE`
The full CORE wordset is present.

`true constant CORE-EXT          \ -- TRUE`
The full CORE-EXT wordset is present.

`false constant FLOORED          \ -- flag`
The standard division operators use symmetric (normal) division.

`#255 constant MAX-CHAR          \ -- u ; max value of char`
Characters are 8 bit units.

`: MAX-D         \ -- d`
Maximum positive value of a double number.

`: MAX-N         \ -- n`
Maximum positive value of a single signed number.

`: MAX-U         \ -- u ; max size unsigned number`
Maximum value of a single unsigned number.

`: MAX-UD        \ -- u ; max size unsigned double`
Maximum value of a double unsigned number.

`rp-size cell / constant RETURN-STACK-CELLS      \ -- n`
Maximum size of the return stack (in cells).

`sp-size cell / constant STACK-CELLS     \ -- n`
Maximum size of the data stack (in cells).

`true constant EXCEPTION         \ -- TRUE`
EXCEPTION word-set is present.

`true constant EXCEPTION-EXT     \ -- TRUE`
EXCEPTION EXT word-set is present.

10.5.2 User words

`' environment >body @ constant environment-wordlist     \ -- wid`
The wid used by `ENVIRONMENT?` for look ups. You can add your own queries to this wordlist.

`: ENVIRONMENT?  \ c-addr u -- false | i*x true                 6.1.1345`
The text string c-addr/u is of a keyword from ANS 3.2.6 Environmental queries or the optional word sets to be checked for correspondence with an attribute of the present environment. If the system treats the attribute as unknown, the returned flag is false; otherwise, the flag is true and the i*x returned is of the type specified in the table for the attribute queried.

`: [environment?]        \ "string" -- false | i*x true`
As `ENVIRONMENT?` but is IMMEDIATE and takes the string from the input stream.

`: .environment  \ --`
Display a list of queries.

10.6 Automatic build numbering

The build numbering system allows you to generate a string in the system which can be used for displaying version information.

The system relies on a file (normally called *BUILD.NO*) which holds the complete build version string. The string can consist of any characters, e.g "Version 1.00.0034". The contents of the file can be placed as a counted string in the dictionary by `BUILD$,`. After successful compilation of your application, `UPDATE-BUILD` will update the build number file by treating **all** the digits in the build string as a single number to be incremented.)

`: Make-Build    \ buffer --`
Read the contents of the build number file and place as a counted string in the application defined buffer for later use.

`: Build$,       \ --`
Read the contents of the build number file and place as a counted string at `HERE`. `ALLOT` the required space.

`: Date$,        \ --`
Compile date as counted string.

`: Time$,        \ --`
Compile time as counted string

`: DateTime$,    \ --`
Compile date and time as counted string

`: Set-BuildFile \ c-addr u --`

Set the build number file.

```
: BuildFile     \ -- ; Buildfile <filename>
```

Use GetPathSpec to parse a filename from the input stream, and make it the current build number file.

```
: Update-Build  \ --
```

Update the contents of the build number file ready for the next build.
The following example, defines which file to use, loads the text into a buffer, and finally updates the build text. By placing `Update-Build` last in your load file, your build number file will only be updated for each successful build.

```
s" MyBuild.no" Set-Buildfile    \ set file to use
#256 buffer: MyVersion$  \ -- caddr
  MyVersion$ make-build         \ load version string
...
update-build                    \ put this last in load file
```

10.7 PDF help system

MPE documentation is produced by using *DocGen* to produce an indexed PDF file. The PDF help system parses the index file produced by *pdftex* to display the relevant page of the PDF manual. To display a particular line in a PDF file requires the following incantation for Adobe Reader v7 and beyond:

```
<reader> /A "page=n=OpenActions" "<pdffile>"
```

The page number is the PDF file page number, not the page number in the document section. For example, to display page 10 on a Windows PC, use:

```
"C:\Program Files\Adobe\Reader 8.0\Reader\AcroRd32.exe"
/A "page=10=OpenActions" "%h%.pdf"
"C:\Products\VfxForth.dev\Sources\Manual\PDFs\VfxWin.pdf"
```

For VFX Forth for Windows, you can use the menu item *Option -> Set PDF help ...* for the configuration.

The page number is extracted from the index file *VfxWin.vix*, from which this example comes:

```
\initial {A}
\entry {\code {abell}}{11}
\entry {\code {abl}}{12}
\entry {\code {abort}}{15, 200}
\entry {\code {abort"}}{200}
```

The file is parsed for the entry containing the word name, the page number is extracted, and the file page is displayed. The index file is derived from the *.fns* file produced by *pdftex*.

The source code is in *Lib\PDFhelp.fth*.

```
TextMacro: p    \ -- $text
```

Defines the page number macro *p*.

```
TextMacro: h    \ -- $text
```

Define the help file macro *h*.

```
#256 constant /Help$    \ -- n
```

Size of the command and base string buffers.

```
/Help$ buffer: HelpCmd$ \ -- addr
```

Holds the pathname and command line of the PDF viewer as a counted string. In the command line, the page number is supplied by the text macro *%p%*, and the base help file path/name with no extension is supplied by the text macro *%h%*. This string may include other macros. For Acrobat Reader under Windows, we must use the full reader pathname; we cannot use an association. The default string is

```
"<reader>" /A \qpage=%p%=OpenActions\q %h%.pdf
```

where <*reader*> is

```
"C:\Program Files\Adobe\Reader 8.0\Reader\AcroRd32.exe"
```

If you change the settings, use `S\"` as you may need to have double-quotes characters in the string around path names that include spaces.

An alternative PDF viewer, which behaves better as a help file viewer and is **much** faster is the free Foxit Reader from

```
http://www.foxitsoftware.com/
```

The required command strings for *Foxit Reader* are

Up to v4

```
"<path>\Foxit Reader.exe" "%h%.pdf" -n %p%
```

From v5.0

```
"<path>\Foxit Reader.exe" /A "page=%p%" "%h%.pdf"
```

The rules for the *Foxit Reader* command line need to be checked with every new release!

Users have also suggested Sumatra, Nitro and PDF-XChange among many others. MPE has no position on this - it's a matter of personal preference. Just check the manual for the command line incantation!

For some Linux systems, e.g. Ubuntu, *xpdf* is installed by default. The VFX defaults are

```
s\" xpdf %h%.pdf %p% &" HelpCmd$ place
s" /usr/share/VfxForth/PDFs/VfxLin" HelpBase$ place
#17 HelpPage0 !
```

Some Linux distributions, e.g. Debian, require shared documentation files to be compressed. Type:

```
locate VfxLin.pdf
```

To see where VfxLin.pdf has been installed, and to see what extension, e.g. *.pdf.gz* is in use.

For OS X, the default PDF viewer is *Preview*. It can be run from the command line using:

```
open -a Preview filename.pdf
```

However, going to a page number is undocumented. The best solution we have found is to install the *Skim* package from:

```
http://skim-app.sourceforge.net/
```

Skim can be run using the supplied executable script *Bin/skimpage.scpt*. This is run in the form:

```
skimpage.scpt "<file>" <pageno>
```

After copying the script file to a suitable directory such as */usr/bin* (done by the install script), a suitable setup is:

```
s\" skimpage.scpt \q%h%.pdf\q %p%" HelpCmd$ place
s" ~/VfxForth/Doc/VfxOsx" HelpBase$ place
#14 HelpPage0 !
```

```
/Help$ buffer: HelpBase$        \ -- addr
```

Contains a full path to the directory containing the help and index files, plus the base file name with no extension. This string may include macros. A counted string, e.g for Windows

```
%LOAD_PATH%\..\doc\VfxMan
```

and for Linux

```
%LOAD_PATH%/../doc/VfxLin
```

and for OS X

```
%LOAD_PATH%/../doc/VfxOsx
```

```
variable HelpPage0      \ -- addr
```

Holds the offset to be added to the index page number to convert it to a PDF page number. This value may change according to the manual version, so it is extracted from the index file.

```
0 value DebugHelp?      \ -- flag
```

Set this non-zero if you are having trouble setting up the help system. The command line will be displayed.

```
: $Help          \ caddr len -- ior ; 0=success
```

Run the help file system using *caddr/len* as the search key.

```
: Help           \ "<word>" -- ; e.g. HELP dup
```

Get help on the given word name, e.g.

```
help locate
```

Unlike `LOCATE`, `HELP` does not require the word to be present in the Forth dictionary and current search order, it only requires that a word have an index entry in the PDF manual.

```
: PDFLoadCfg    \ --
```

Load the PDF help configuration from the INI file. Linux and OS X only.

```
: PDFSaveCfg    \ --
```

Load the PDF help configuration from the INI file. Linux and OS X only.

10.8 INI files

If you are upgrading from a system installed before March 2012, you may/will need to change how your application specifies its INI files.

The INI file mechanism used by Windows is also available for other operating systems. This allows us to use the same configuration file mechanism for all operating systems that support shared libraries (not in VFX Forth for DOS yet).

The code accesses a derivative of the iniParser v3.0b shared library published by Nicholas Devillard at *http://ndevilla.free.fr/iniparser/*, where the latest version may be found. Note that the MPE versions differ from this version, but are upward compatible. We have submitted our changes to the author. A binary copy of the library can be found in the *Bin* folder. You are free to release this with your applications.

The full sources for iniParser as used by MPE are in the directories *<VFX>\Tools\iniparser3.0b\src* and *<VFX>\Tools\iniparser3.0b\src.win*. The relevant shared library (.so or .dll) is in the *Bin* folder and is copied to the *Windows\System32* or */user/lib* directory during installation.

The private profile file mechanism used by VFX Forth for Windows before version 4.20 may be found in *<VFX>Lib\Win32\Profile.fth*. The old mechanism is not portable between operating systems and is much slower. We strongly recommend that you convert any existing code that uses the `PROFILE::xxx` words to use the new mechanism.

The follwing example shows how INI files are used. A later section describes the words in detail.

```
: SaveSome        \ --
  s" %IniDir%\UserIde.ini" Ini.Open 0= if
    S" Options" Ini.Section
    s" FontSet" FontSet? Ini.WriteInt
    s" LogFont" lf[ LOGFONT Ini.WriteMem
    s" Editor"  szEditor zcount Ini.WriteStr
    Ini.Close
  endif
;

: LoadSome   \ --
  s" %IniDir%\UserIde.ini" Ini.Open 0= if
    S" Options" Ini.Section
    s" FontSet" 0 Ini.ReadInt dup -> FontSet? if
      s" LogFont" lf[ LOGFONT Ini.ReadMem
      lf[ CreateFontIndirect -> hConsoleFont
    endif
    s" Editor" szEditor MAX_PATH zNull zcount Ini.ReadZStr
    Ini.Close
  endif
;
```

The main things to note are:

- The INI file must be opened before use.
- Sections are the parts of the INI file delimited by `[name]`. Sections contain key/value pairs in the form `key=value`. The value portion is represented as text.
- Your code can use a key as a flag to determine how others are handled.
- The INI file must be closed, otherwise changes will not be written back.

10.8.1 Shared library interface

`Library: libmpeparser.so.0`
The library reference.

`: IniLib        \ -- addr|0`
Used to determine if the library is available and to isolate the host-dependent library name from the rest of the code.

`Library: libarmmpeparser.so.0`
The library reference.

`: IniLib        \ -- addr|0`
Used to determine if the library is available and to isolate the host-dependent library name from the rest of the code.

`Extern: int iniparser_getnsec( void * dict );`
This function returns the number of sections found in a dictionary. The test to recognize sections is done on the string stored in the dictionary: a section name is given as "section" whereas a key is stored as "section:key", thus the test looks for entries that do not contain a colon. This function returns -1 in case of error.

`Extern: char * iniparser_getsecname( void * dict, int n );`
This function locates the n-th section in a dictionary and returns its name as a pointer to a string statically allocated inside the dictionary. Do not free or modify the returned string! This function returns NULL in case of error.

`Extern: void iniparser_dump_ini( void * dict, void * file );`

This function dumps a given dictionary into a loadable ini file. It is Ok to specify *stderr* or *stdout* as output files.

`Extern: void iniparser_dump( void * dict, void * file );`

This function prints out the contents of a dictionary, one element by line, onto the provided file pointer. It is OK to specify *stderr* or *stdout* as output files. This function is meant for debugging purposes mostly.

`Extern: char * iniparser_getstring( void * dict, const char * key, char * def );`

This function queries a dictionary for a key. A key as read from an ini file is given as "section:key". If the key cannot be found, the pointer passed as 'def' is returned. The returned char pointer is pointing to a string allocated in the dictionary, do not free or modify it.

`Extern: int iniparser_getint( void * dict, const char * key, int notfound );`

This function queries a dictionary for a key. A key as read from an ini file is given as "section:key". If the key cannot be found, the notfound value is returned. Supported values for integers include the usual C notation so decimal, octal (starting with 0) and hexadecimal (starting with 0x) are supported. Examples:

```
- "42"      ->  42
- "042"     ->  34 (octal -> decimal)
- "0x42"    ->  66 (hexa  -> decimal)
```

Warning: the conversion may overflow in various ways. Conversion is totally outsourced to strtol(), see the associated man page for overflow handling.

`Extern: int iniparser_getboolean( void * dict, const char * key, int notfound );`

This function queries a dictionary for a key. A key as read from an ini file is given as "section:key". If the key cannot be found, the notfound value is returned. A true boolean is found if one of the following is matched:

```
- A string starting with 'y'
- A string starting with 'Y'
- A string starting with 't'
- A string starting with 'T'
- A string starting with '1'
```

A false boolean is found if one of the following is matched:

```
- A string starting with 'n'
- A string starting with 'N'
- A string starting with 'f'
- A string starting with 'F'
- A string starting with '0'
```

The notfound value returned if no boolean is identified, does not necessarily have to be 0 or 1.

`Extern: int iniparser_set( void * dict, char * entry, char * val );`

If the given entry can be found in the dictionary, it is modified to contain the provided value. If it cannot be found, -1 is returned. It is Ok to set val to NULL.

`Extern: void iniparser_unset( void * dict, char * entry );`
If the given entry can be found, it is deleted from the dictionary.

`Extern: int iniparser_find_entry( void * dict, char * entry );`
Finds out if a given entry exists in the dictionary. Since sections are stored as keys with NULL associated values, this is the only way of querying for the presence of sections in a dictionary.

`Extern: void * iniparser_load( const char * ininame );`
This is the parser for ini files. This function is called, providing the name of the file to be read. It returns an ini dictionary object that should not be accessed directly, but through accessor functions instead. The returned dictionary must be freed using **iniparser_freedict()**.

`Extern: int iniparser_save( void * d, char * ininame );`
Saves a dictionary object. This is just a wrapper around iniparser_dump_ini() to provide insulation between the caller and the file system for languages and operating systems which do not expose the libc library. The returned error code is 0 for a successful operation. You still need to call `iniparser_freedict` below.

`Extern: void iniparser_freedict( void * dict );`
Free all memory associated to an ini dictionary. It is mandatory to call this function before the dictionary object gets out of the current context.

10.8.2 Tools

These tools are in the `SYSTEM` vocabulary and may change from version to version. If all you are interested in is using the MPE library interface API, skip this section.

`0 value IniSrcFile      \ -- addr`
Holds the currently loaded INI source file pathname as a zero-terminated string.

`0 value IniDestFile     \ -- addr`
Holds the currently loaded INI destination file pathname as a zero-terminated string.

`0 value IniSection      \ -- addr`
Holds the current section name as a zero-terminated string.

`0 value IniKey          \ -- addr`
Holds the current key as a zero-terminated string.

`0 value IniData         \ -- addr`
Holds the current write data as a zero-terminated string,

`0 value IniDefault      \ -- addr`
Holds the current default as a zero-terminated string

`0 value IniScratch      \ -- addr`
Holds the current scratch buffer for processing quote marks.

`0 value IniDict         \ -- addr`
Holds the current dictionary pointer.

`: IniAlloc      \ ptr -- ior`
Allocate `/IniBuff` bytes and place the buffer address at *ptr*. The first bye of *ptr* is set to zero.

`: IniFree       \ ptr --`
Free buffer memory allocated by us.

`: InitIniBuffs  \ -- ior`
Initialise all the buffers and pointers.

`: TermIniBuffs  \ --`
Free all the buffers and clear pointer.

`: +DoubleQ      \ z$1 -- z$2`
Convert double quote characters to pairs of double quote characters.

`: -DoubleQ      \ z$1 -- z$2`
Convert pairs of double quote characters to single double quote characters.

`: >IniName      \ caddr len dest --`
Copy name to zero terminated string.

`: >IniString    \ caddr len dest --`
Copy string to zero terminated string. If the last character is a '\', add a dummy comment `" ; "`.

`: WriteIniFile  \ --`
Write the current INI dictionary to the INI file.

`: FormIniKey    \ caddr u --`
Form the key string from the current section name and the given key. The key string is of the form `"<section:<key>"`.

`: setIniString  \ dict entry val --`
Calls **iniparser_set()** and marks the INI file as changed.

`: IniExists     \ caddr len --`
If the file does not exist create an empty one.

`: czplace       \ caddr len dest`
Store the string *caddr/len* as a counted and zero-terminated string at *dest*. The strings must not overlap.

`: nib>hex       \ 4b -- char`
Convert a nibble to a hex character.

`: Mem>Hex       \ caddr len zdest --`
Generate an ASCII hex representation of the memory block as a zero terminated string at *zdest*. The length *len* of the memory block must be less than 128 bytes.

`: Hex>Nib       \ char -- 4b`
Convert a hex character to a nibble.

`: Hex>Mem       \ zsrc caddr len --`
Convert the zero-terminated ASCII HEX string at *zsrc* to its memory representation in the buffer *caddr/len*.

10.8.3 Using the library

The code here is not thread safe, you so may need a semaphore from open to close. Some data is held in global variables/buffers.

`: Ini.Open      \ caddr len -- ior`
Define and load the Ini file, returning zero on success. Sets the destination file to be the same as the loaded file. Macros in the file name are expanded.

`: Ini.Dest      \ caddr len --`

Set the destination file. This **must** be done after `Ini.Open` and before `Ini.Close`.

```
: Ini.Close      \ --
```

If the dictionary has been changed, write it out to the destination file.

```
: Ini.Section    \ caddr len --
```

Set the current section name.

```
: Ini.Section?  \ caddr u -- flag
```

Given a section name, make it current, and return true if it exists.

```
: Ini.WriteSection      \ caddr u --
```

Make the given section current, and write it to the dictionary.

```
: Ini.ReadStr   \ c-addr1 u1 c-addr2 u2 c-addr3 u3 --
```

Read a string value under key *c-addr1/u1* and return it in the result buffer specified by *c-addr2/u2*. If the key couldn't be read then the default string *c-addr3/u3* is placed in the result buffer. Note that the returned string is placed in the result buffer as a zero terminated **and** counted string.

```
: Ini.ReadZStr  \ c-addr1 u1 c-addr2 u2 c-addr3 u3 --
```

Read a string value under key *c-addr1/u1* and return it in the result buffer specified by *c-addr2/u2*. If the key couldn't be read then the default string *c-addr3/u3* is placed in the result buffer. Note that the returned string is placed in the result buffer as a zero terminated string.

```
: Ini.ReadInt   \ c-addr1 u1 default -- value
```

Attempt to read an integer value from key *c-addr1/u1* and return it. If the key couldn't be read then the default is returned.

```
: Ini.ReadBool  \ c-addr1 u1 defbool -- bool
```

Attempt to read an boolean value from key *c-addr1/u1* and return it. If the key couldn't be read then the default is returned.

```
: Ini.ReadMem   \ c-addr1 u1 c-addr2 u2 --
```

Read a memory block with key *c-addr1/u1* and return it in the result buffer specified by *c-addr2/u2*. If the key couldn't be read then the result buffer is filled with zero bytes. *u2* must be less than 128 bytes.

```
: Ini.WriteStr  \ c-addr1 u1 c-addr2 u2 --
```

Write a string value attached to a key, into the currently named section of the current INI file. *C-addr1/u1* describes the name of the key to place the entry under, and *C-addr2/u2* the string to write. If the key already exists it is updated.

```
: Ini.WriteZStr \ c-addr1 u1 caddrz --
```

Write a zero-terminated string value attached to a key, into the current section of the INI file. *C-addr1/u1* describes the name of the key to place the entry under, and *caddr*\ the string to write. If the key already exists it is updated.

```
: Ini.WriteInt  \ c-addr1 u1 u2 --
```

Write a string value corresponding to decimal text for *u2* to the key *C-addr1/u1* in the current section of the current dictionary.

```
: Ini.WriteBool \ c-addr1 u1 bool --
```

Write a string value corresponding to decimal text for *bool* to the key *C-addr1/u1* in the current section of the current dictionary.

```
: Ini.WriteMem  \ c-addr1 u1 c-addr2 u2 --
```

Write a memory block attached to a key, into the currently named section of the current INI file. *C-addr1/u1* describes the name of the key to place the entry under, and *C-addr2/u2* the memory block to write. If the key already exists it is updated.

`: Ini.DeleteKey \ c-addr1 u1 --`

Delete a key entry completely from the current section. *C-addr1/u1* is the name of the key.

10.8.4 Operating system generics

`#256 buffer: IniFile$   \ -- addr`

A 256 byte buffer for the expanded INI file path name, which may include macros. The path name is stored as a counted string. This buffer is initialised at startup from the three components below.

`#256 buffer: IniDir$    \ -- addr`

A 256 byte buffer for the expanded INI file directory name, which may include macros. The path is stored as a counted string. This buffer is initialised at startup

`256 buffer: AppSupp$    \ -- addr`

The directory where application support files go, held as a counted string This directory must already exist. May contain macros. You can change this for your own application.

`64 buffer: AppSuppDir$  \ -- addr`

The directory in `AppSupp$` in which the INI file is placed, held as a counted string. The directory will be created if it does not already exist. This string may be null, or may define one or two levels of directory. You can change this for your own application.

`64 buffer: AppSuppIni$  \ -- addr`

The name of the INI file in `AppSuppDir$`, held as a counted string. By default, the file is *VfxForth.ini*. You can change this for your own application.

`256 buffer: PrevIni$    \ -- addr`

Holds the full pathname of a default file copied to the INI file if it does not exist. May contain macros. You can change this for your own application.

`-1 value GenINI?        \ -- flag ; true to generate INI files`

If this `VALUE` is set true (the default condition) .INI files will be generated for VFX Forth and applications when the application performs `BYE`. Such files will also be loaded when VFX Forth or an application is executed.

`defer CheckSysIni       \ --`

This word is run at cold start before any INI file is loaded. It should provide an INI file if one is needed and does not exist.

`1 value IniParserModes  \ -- modes`

If you need only one INI file that could be in the directory from which the application is loaded, set this to zero.

`TextMacro: IniFile`

A text macro that returns the INI file name after macro expansion at startup.

`TextMacro: IniDir`

A text macro that returns the INI file directory after macro expansion at startup.

`: -ini-exec     \ --`

When used on the command line in lower case, `-ini-exec` causes the INI file to be loaded from `PrevIni$`, which is usually in the executable directory.

`: (CheckSysIni) \ --`

Creates the INI file directory if required and sets up the INI file macros. This is the default action of `CheckSysIni`.

10.8.5 Operating system specifics

Setting the INI files has changed. You must now set up four strings rather than two, and `IniFile$` is set at startup, and not by you. The defaults are shown below for three operating systems.

These changes were required by changes in the Windows security system, the desire to fit in "well" with Unix-derived systems, and the requirement for multiple application-specific data files. See the `IniDir` macro in particular.

Windows

By default, the INI file is *%$AppLocal%\MPE\VfxForth\VfxForth.ini*. If the directory or file does not exist, as may happen after installation, the directory is created and/or a default file is copied.

```
s" %$AppLocal%" AppSupp$ place              \ system dir
s" MPE\VfxForth" AppSuppDir$ place          \ our dir
s" VfxForth.ini" AppSuppIni$ place          \ file
s" %load_path%\VfxForth.ini" PrevIni$ place   \ default/previous
```

Mac OS X

By default, the INI file is *%$home%/Library/Application Support/VfxForth/VfxForth.ini*. If the directory or file does not exist, as may happen after installation, the directory is created and/or a default file is copied.

```
s" %$home%/Library/Application Support" AppSupp$ place
s" VfxForth" AppSuppDir$ place
s" VfxForth.ini" AppSuppIni$ place
s" %$home%/.VfxForth.ini" PrevIni$ place
```

Linux

By default, the INI file is *%$home%/.VfxForth/VfxForth.ini*. If the directory or file does not exist, as may happen after installation, the directory is created and/or a default file is copied.

```
s" %$home%" AppSupp$ place
s" .VfxForth" AppSuppDir$ place
s" VfxForth.ini" AppSuppIni$ place
s" %$home%/.VfxForth.ini" PrevIni$ place
```

10.8.6 System initialisation chains

These chains are used for configurations options which are preserved when VFX Forth closes down, and are reloaded when it starts. Note that because of the position in the cold and exit chains, you must be careful that you still have the configuration data. For example, if a window is closed before configuration save, its position may have to be saved in a buffer rather than reading it directly from the window. Similarly, when the configuration information is restored, the window may/will not be open yet.

If you want to read and write directly from live information, it is more reliable to provide full handlers in your application code. However, there will be a time penalty because the INI file is repeatedly opened and closed.

`variable IniLoadChain   \ -- addr`
The anchor for the initialisation load sequence.

`variable IniSaveChain   \ -- addr`
The anchor for the initialisation save sequence.

`: AtIniLoad     \ xt --`
The given word is run during the INI load sequence.

`: AtIniSave     \ xt --`
The given word is run during the INI save sequence.
The words run must have no net stack action, and behave according to the rules of `ExecChain`.

`: LoadSysIni    \ --`
Open the INI file specified by `iniFile$`, run the `IniLoadChain`, and close the file. Run in the cold chain. No action is taken if `GenINI?` is zero.

`: SaveSysIni    \ --`
Open the INI file specified by `iniFile$`, run the `IniSaveChain`, and close/save the file. Run in the exit chain. No action is taken if `GenINI?` is zero.

10.9 Converting from the previous mechanism

By design, there is an almost one to one correspondence between the words in the profile and INI mechanisms. There are two major differences.

- The new code rarely returns error codes as we and you nearly always `DROP`ped them.
- You `must` use `Ini.Close` after you have finished with the INI file. The data is in memory, and is only flushed at close.

For examples of use, see `LoadUserIde` and `SaveUserIde` in *Studio\XTB.FTH* for the Windows versions.

10.10 Switch chains

10.10.1 Introduction

Switch chains provide a mechanism for generating extensible chains similar to the CASE ... OF ... ENDOF ... ENDCASE control structure, except that the user may extend the chain at any time. These chains are of particular use when defining winprocs whose action may need to be adjusted or extended after the chain itself has been defined.

The following example shows how to define a simple chain that translates numbers to text. At a later date, translations in Italian are added.

Define some words which will be executed by the chain.

```
: one   ." one"  ;
: two   ." two"  ;
: three ." three" ;
: many  . ." more" ;
```

The following definition defines a switch called NUMBERS which executes ONE when 1 is the selector, TWO if 2 is the selector, or MANY if any other number is the selector. Note that MANY must consume the selector. The word RUNS associates a word with the given selector.

```
[switch numbers many
  1 runs one
  2 runs two
switch]
cr 1 numbers
cr 5 numbers
```

The next piece of code extends the NUMBERS switch chain, and demonstrates the use of RUN: to define an action without giving it a name.

```
[+switch numbers
  3 runs three
  4 run: ." four" ;
  5 run: ." five" ;
switch]
cr 1 numbers
cr 5 numbers
cr 8 numbers
```

The following portion of this example demonstrates how selectors are overridden by the last action defined. Although an action has already been defined for selectors 1 and 2, if another action is defined, it will be found before the old ones, and so the action will be performed.

```
[+switch numbers
  1 run: ." uno" ;
  2 run: ." due" ;
switch]
cr 1 numbers
cr 2 numbers
cr 3 numbers
cr 5 numbers
cr 8 numbers
```

10.10.2 Switches glossary

`code switch     \ i*x id switchhead -- j*x`

Given an id and the head of a switch chain, SWITCH will perform the action of the given id if it is found, otherwise it will perform the default action, passing id to that action.

`: [switch       \ "default" -- head ; i*x id -- j*x`

Builds a new named switch list with a default action. Use in the form: [SWITCH <name> <default.action> where <default.action> must consume the selector id.

: [+switch \ "head" -- head ; to extend an existing switch
Used in the form [+SWITCH <switch> to extend an existing switch chain.

: run: \ head id -- head ; add nameless action to switch
Used in the form <id> RUN: <words> ; to define a nameless action in a switch chain.

: runs \ head id "word" -- head
Used in the form <id> RUNS <word> to define a named action in a switch chain.

: switch] \ head -- ; finishes a switch chain or extension
Used to finish a [SWITCH <name> <default> or [+SWITCH <name> chain definition.

: .switches \ -- ; lists defined switches
Lists all the defined switch chains.

: InSwitch? \ id xt -- flag
Returns true if the **id** is in the switch chain given by its **xt**.

10.11 First-In First-Out Queues

VFX Forth contains a set of words for managing queues. These queues are allocated from the system heap.

STRUCT FIFO \ -- len
A structure which defines the internal format of the fifo. To create a new FIFO you must create an instance of the structure. The instance pointer (address of structure) is then used as an identifier for subsequent operations.

: InitialiseFIFO \ *FIFO size -- ior
Initialise a FIFO with a maximum buffer 'size' bytes long. the IOR is 0 for success and non-zero for memory allocation failed.

: FreeFIFO \ *FIFO -- ior
Destroy FIFO. Memory is released back into the heap.

: FIFO? \ *FIFO -- n ; Return # bytes used in fifo
Return the number of storage bytes in use within a fifo.

: >FIFO(b) \ BYTE *FIFO -- ior
Add a byte to the FIFO queue. IOR is 0 for success.

: FIFO>(b) \ *FIFO -- byte ior
Remove next byte from a FIFO. IOR is 0 for success.

10.12 Random numbers

The random number system in VFX Forth is based around the DEFERred word RANDOM (see below). We have found that a single random number generator (RNG) is not adequate for all applications. Some applications need a particular degree of randomness, others require more speed. If you do not like the default RNG, you can install your own.

The implementation uses a seed in the variable *\fo{RandSeed), which is set to some time value at start up. The default implementation is in the SYSTEM vocabulary.

variable RandSeed \ -- addr
Seed value used by the RNG. Set to TICKS at start up.

defer RANDOM \ -- u

Generate a 32 bit random number.

`: CHOOSE          \ n1 -- n2`

Generate a random number *\i{n2) in the range 0..n1-1. The algorithm is from Paul Mennen, 1991 .

10.13 Long Strings

In order to extract very long strings from the source code, the word PARSE/L is provided. Support is also provided for counted strings with a 16 bit count. These words allow long strings such as those required for internationalisation to be generated without the restrictions of counted strings that use a character-sized count.

The contents of this section are subject to change until the ANS Forth committee reaches a conclusion about internationalisation issues. An implementation of the system descibed in the paper on the MPE web site may be found in the file %LIB%\INTERNATIONAL.FTH.

`: parse/l         \ char -- c-addr len ; like PARSE over lines`

Parse the next token from the terminal input buffer using <char> as the delimiter. The text up to the delimiter is returned as a c-addr u string. PARSE/L does not skip leading delimiters. In order to support long strings, PARSE/L can operate over multiple lines of input and line terminators are not included in the text. The string returned by PARSE/L remains in a single global buffer until the next invocation of PARSE/L. PARSE/L is designed for use at compile time and is not thread-safe or winproc-safe.

`: wcount           \ addr1 -- addr2 len`

Given the address of a word-counted string in memory WCOUNT will return the address of the first character and the length in characters of the string.

`: (W")            \ -- waddr u ; step over caller's in line string`

Returns the address and length of inline 16-bit word-counted string. Steps over inline text.

`: ((W"))          \ -- waddr u ; dangerous factor!`

A factor provided for the generation of long string actions that have to step over an inline string. For example, to define W." which uses a long string, you might compile (W.") and then use W", to compile the inline string. The definition of (W.") then might be:

```
: (W.")     \ --
  ((W")) type
;
```

`: wAppend         \ c-addr u $dest --`

Add string c-addr u to the end of word-counted string $dest.

`: (w$+)          \ c-addr u $dest -- ; SFP001`

Add string c-addr u to the end of word-counted string $dest.

`: w$+             \ $src $dest -- ; add $SRC to end of $DEST`

Add word-counted string $src to the end of word-counted string $dest.

`: w",             \ -- ; compile a word counted string`

multiline version of ",. Interprets multiline text and lays down inline string string with 16 bit count and 16 bit zero termination.

10.14 Command Line parser

VFX Forth includes code for handling the Linux command line. You can access the command line using Forth versions of `ARGV[]` and `ARGC` as in C. At start up, VFX Forth attempts to recreate the original command line using **argc** and the **argv**[] strings so that the command line can be treated as a sequence of Forth commands.

The attempt to recreate the command line is only partially successful because the C parser uses double-quotes characters to denote literal strings. These are passed to the application as a single string with the double-quotes characters removed. For example:

```
vfxlin s" foo   bar" type
```

will give a command line error. You can escape the double-quotes characters in the usual C manner, but this will not preserve the spaces.

```
vfxlin s\" foo   bar\" type
```

Most Linux system shells, including Bash, allow you to use single-quotes characters for literal strings.

```
vfxlin 's" foo   bar" type'
```

`: argc  \ -- u`

The number of defined arguments.

`: argv[                  \ n -- pointer|0`

Given an index of 0..argc-1 return a pointer to the command line token's zero-terminated string. `0 argv[` returns the executable's name. If the argument does not exist, the pointer is zero.

`: CommandLine   \ -- c-addr len`

Return the system command line. This is recreated from the data passed to VFX Forth and may not be exact.

11 Linux specific tools

The code described here is specific to VFX Forth for Linux. Do not rely on any of the words documented here being present in any other VFX Forth implementation.

11.1 Shell operations

The VFX Forth console supports a number of command shell operations.

11.1.1 Primitives

The words in this section are used to build the tools.

`: csplit        \ caddr len char -- raddr rlen laddr llen`

Extract a substring at the start of *caddr/len*, returning the string *raddr/rlen* which includes *char* (if found) and the string *laddr/llen* which contains the text to left of *char*. If the string does not contain the character, *raddr* is *caddr+len* and *rlen*=0.

`: xtype         \ caddr len --`

As `TYPE`, but LF characters cause a `CR`. This factor copes with some user-written generic I/O devices that do not implement `TYPE` correctly.

`: >pShell       \ z$ -- ior`

Execute the given zero-terminated string as a shell command, write any output to the current output device, and return the result code from the **popen()** call. This word provides consistent action regardless of whether operation is running in a console or is detached. This is the default action of `(>xShell)` below.

`defer >xShell   \ z$ -- ior`

Execute the given zero-terminated string as a shell command, and return the result code from the relevant system call such that zero=success. Most words that cause shell actions use `>xShell` as a primitive. To use a raw **system** call instead as the action use:

```
assign ssystem to-do >xShell
```

`: (>Shell)      \ z$ -- ior`

Execute the given zero-terminated string as a shell command, and return the result code from `>xShell` above.

`: >system       \ z$ -- ior`

Execute the given zero-terminated string as a shell command using the **system()** API call, and return the result code.

`: >Shell        \ z$ --`

Execute the given zero-terminated string as a shell command using `(>Shell)` above. Output from the command is written to the current output device.

`: ShellCmd      \ caddr len --`

Execute the given *caddr/len* string as a shell command.

`: ShellLine     \ caddr len --`

Execute the given counted string as a shell command. Before execution, the remainder of the input line is added to the given string.

`: $shell        \ cmd$ tail$ --`

Take the command and tail counted strings and execute them as a shell command using **system()**.

```
: $linux        \ cmd$ tail$ --
```
A synonym for `$shell`.

11.1.2 Command operations

```
: sh            \ -- ; "command"
```
Ask the host operating system to execute the supplied command line.

```
: ls            \ -- ; "[spec]"
```
Display file information based on the supplied specification.

```
: dir           \ -- ; "[spec]"
```
Display file information based on the supplied specification. As `LS` but with colouring.

```
: makedir         \ -- ; "name"
```
Create a new subdirectory from the current working one. This word has been renamed to avoid a name conflict with the system **mkdir()** API.

```
: deldir         \ -- ; "name"
```
Remove a specified subdirectory. You can only remove an empty directory. This word has been renamed to avoid a name conflict with the system **rmdir()** API.

```
: rm            \ -- ; "spec"
```
Delete a single file or group of files as described by the given file specification. The wildcard '*' may also be used.

```
: cat           \ -- ; "spec"
```
Perform an ASCII display of a file or group of files. No filtering of the data is performed. This command should not be used to list binary files.

```
: pwd           \ --
```
Display the currently active working directory using the shell **pwd** command.

```
: cd            \ -- ; ["name"]
```
Attempt to change current working directory either as an offset from the current directory or as a complete path. The wildcard '*' can be used to match the first directory. If there is no tail, `CD` displays the current directory. No shell functions are used.

11.2 Linux signal handling

11.2.1 Structures

Signal numbers See /usr/src/linux-2.6.8-24/include/asm-i386/sigcontext.h

```
#define SIGHUP           1
#define SIGINT           2
#define SIGQUIT          3
#define SIGILL           4
#define SIGTRAP          5
#define SIGABRT          6
#define SIGIOT           6
#define SIGBUS           7
#define SIGFPE           8
#define SIGKILL          9
#define SIGUSR1         10
#define SIGSEGV         11
#define SIGUSR2         12
```

```
#define SIGPIPE         13
#define SIGALRM         14
#define SIGTERM         15
#define SIGSTKFLT       16
#define SIGCHLD         17
#define SIGCONT         18
#define SIGSTOP         19
#define SIGTSTP         20
#define SIGTTIN         21
#define SIGTTOU         22
#define SIGURG          23
#define SIGXCPU         24
#define SIGXFSZ         25
#define SIGVTALRM       26
#define SIGPROF         27
#define SIGWINCH        28
#define SIGIO           29
#define SIGPOLL         SIGIO
#define SIGPWR          30
#define SIGSYS          31
#define SIGUNUSED       31
```

`struct /fpstate \ -- len`

Regular FPU environment. See */usr/src/linux-2.6.8-24/include/asm-i386/sigcontext.h*.

`struct /_libc_fpstate   \ -- len`

The libc FPU environment. See */usr/src/linux-2.6.8-24/include/asm-i386/ucontext.h*.

`struct /sigcontext      \ -- len`

CPU **sigcontext** structure. See */usr/src/linux-2.6.8-24/include/asm-i386/sigcontext.h*. Note that this is **not** the same as a **ucontext** structure.

11.3 Stack description structure.

`struct /gregset_t       \ -- len`

CPU **gregset_t** structure. See */usr/src/linux-2.6.8-24/include/asm-i386/ucontext.h*.

`struct /mcontext        \ -- len`

System **uncontext** structure returned by signal handlers. for Linux i32 this is the same as the **/sigcontext** structure.

`struct /ucontext        \ -- len`

System **uncontext** structure returned by signal handlers.

`struct /sigaction       \ -- len`

System **sigaction** structure.

11.3.1 Signal handling

The following is the stack structure seen by the **siginfo** signal handler.

```
   |             |
   +-------------+
   | Return code |              8 bytes of code to return to kernel context at
   |             |              end of the handler run via a special system call
   |             |              (now a dummy left as a signature for debuggers
   |             |              as many systems don't allow execution from the
   |             |              stack)
   +-------------+
   | Floating    |              FPU state if FPU has been used by this process
   | point state |
   +-------------+
   | User        |              POSIX extension user context for the signal
   | Context     |          handler.  Includes register contents etc, many
+->|             |              of which are modifiable by the signal handler
|  +-------------+
|  | Siginfo     |              Traditional signal information structure.
|  | structure   |<-+   Duplicates/simplifies some of User Context
|  +-------------+  |
+--| void *      |       |      Pointer to POSIX user context
   +-------------+  |
   | siginfo_t * |--+   Pointer to Siginfo structure
   +-------------+
   | Signum      |              Actual signal number generated
   +-------------+
   | Return      |              Originally this pointed at the stack based
   | address     |              return code above.  Now points directly at the
   |             |              sigreturn syscall gate in the vDSO
   +-------------+
```

`create sigNames \ -- addr`
Holds the signal numbers and names as counted strings.

`: .sigName      \ n --`
Given a signal number, display its name.

`: .RSitem       \ x --`
Display an item retrieved from the faulting return stack.

`: .SigContext   \ sc --`
Display data from the sigcontext structure.

`: SigThrow      \ --`
Runs the O/S THROW action.

`3 0 callback: SigGenTrap        \ signum *siginfo *ucontext --`
Generic trap handler that causes a `-57005 THROW` on return. Callbacks are documented in a separate section of the manual.

`1 value -NestedSigs?    \ -- x`
Set non-zero to cause an exit if a nested signal exception occurs in `(SigGenTrap)` below.

`1 value SigPause?       \ -- x`
Set non-zero to cause a pause when a signal is processed in `(SigGenTrap)` below.

`: (SigGenTrap)  \ signum *siginfo *ucontext --`
Action of `SigGenTrap`. Displays an error message. On return to Linux, a Forth `THROW` will occur.

`: setSignal     \ callback signum --`

Set a signal handler to execute the given callback. The callback must have the stack effect `( signum *siginfo *ucontext -- )`

`: setSigTraps   \ --`

Install the `SigGenTrap` signal handler for signals SIGILL, SIGFPE and SIGSEGV. Performed at startup.

11.4 Error variables

VFX Forth for Linux uses many functions from the **libc** shared library. The thread local error variables are exposed.

`AliasedExtern: errno int * __errno_location( void );`

`errno` is the well known *errno* C thread local variable used by libraries and system calls. Can be read by `@` and written by `!`

`AliasedExtern: h_errno int * __h_errno_location( void );`

`h_errno` is the *h_errno* C thread local variable. Can be read by `@` and written by `!`

11.5 Environment variables

`: ReadEnv       \ naddr1 nlen -- vaddr vlen`

Read the environment variable whose name is given by *naddr/nlen* and return the string. If there is no such variable *vaddr* is `zNull` and *vlen* is zero.

`: WriteEnv      \ vaddr vlen naddr nlen --`

Write the string value *vaddr/vlen* to the environment variable named by *vaddr/vlen*.

`: DelEnv        \ naddr nlen --`

Delete the environment variable *naddr/nlen*.

`: EnvMacro:     \ naddr nlen "<var>" -- ; -- caddr`

Create a text macro called <*var*> that queries the environment variable named by *naddr/nlen* the returned string is a counted string. Use in the form:

```
s" HOME" EnvMacro: $home
```

By convention, environment macro names start with a '$'.

`s" HOME" EnvMacro: $home`

Text macro for the home directory.

11.6 Critical sections

Critical sections are implemented using the standard Linux semaphore structures and calls.

`16 constant /sem_t      \ -- len`

Size of a Linux i32/ARM **sem_t** structure. You can treat this as an opaque type that you do not have to deal with directly. All you have to do is to reserve memory for it, e.g.

```
/sem_t buffer: MyCritSec
```

`: InitCritSec   \ sem --`

Initialise the critical section. This **must** be done before using it.

`: TermCritSec   \ sem --`

Delete the critical section associated with the smaphore. This releases internal Linux data, the `/sem_t` structure is still available but needs to be initialised again before reuse. Nothing should be waiting on the seamphore before calling `TermCritSec`.

`: [CritSec      \ sem --`
Wait until the section is available and lock it. Does not call `PAUSE`.

`: CritSec]      \ sem --`
Unlock the section.

`: CritSec?      \ sem -- u`
Returns the section's counter, where non-zero indicates that it is available, or zero when it is locked. Returns zero on error.
The critical section words use Linux semaphores, which are counted semaphores. Thus when using critical sections you must be careful to match the use of `[CritSec` and

11.7 Millisecond timer

The Linux ticker frequency varies between implementations. The code in this section provides simple tools to return and handle a millisecond ticker.

`: (ticks)       \ -- ms ; return ticks in ms`
Return the system ticker in milliseconds. Treat this as a 32 bit unsigned value that wraps around on overflow.

`: SetTicks      \ --`
Calibrate the Linux ticker and install it as the action of `TICKS`. Performed at start up.

`5 value tickStepMs      \ -- ms`
Minimum interval and granularity used by `tick-ms` below.

`: tick-ms       \ ms --`
Waits for at least *ms* milliseconds. Uses `PAUSE` every `tickStepMs`. This is the default action of `MS`, which is `DEFER`red.

11.8 Time handling

11.9 A structure to mimic the timeval structure for libc

```
  4 field tv_sec
  4 field tv_nsec
end-struct
```

11.10 A structure to mimic the tm structure for libc

`: td>epoch      \ seconds mins hours day month year -- epoch`
Returns the seconds since the start of the epoch. The input time is treated as GMT/UTC.

`: epoch>td      \ epoch -- seconds mins hours day month year`
Converts an epochal second into a GMT/UTC time and date.

11.11 Time and date

These functions rely on the ANS Forth word `TIME&DATE ( -- s m h dd mm yyyy )` and the non-standard `DOW ( -- dow, 0=Sun)` to get the day of the week.

```
create days$    \ -- addr
```
String containing 3 character text for the days of the week.

```
create months   \ -- addr
```
String containing 3 character text for the months.

```
: .dow          \ dow --
```
Display day of week.

```
: .2r           \ n --
```
Display n as a two digit number with leading zeros.

```
: .4r           \ n --
```
Display n as a four digit number with leading zeros.

```
: .Time&Date    \ s m h dd mm yy --
```
Display the system time The format is:

```
hh:mm:ss dd Mmm yyyy
```

```
: .AnsiDate     \ zone --
```
Display the day of week, date and time. If zone is 0 GMT (system time) is displayed, otherwise local time is displayed. The format is:

```
dow, hh:mm:ss dd Mmm yyyy [GMT]
```

11.12 Hardware I/O port access

The following notes are for developers working under x86-32 versions of Linux. Under normal use, direct access to I/O ports is forbidden. However, if you are running with root privilege, you can use the glibc functions **ioperm()** and **iopl()** to enable and disable port access.

```
code pc@        \ port -- b ; read port
```
Read a byte from the hardware control port supplied.

```
code pc!        \ b port -- ; write port
```
Write the supplied byte to the selected hardware control port.

```
code pw@        \ port -- w ; read port
```
Read a 16 bit word from the hardware control port supplied.

```
code pw!        \ w port -- ; write port
```
Write the supplied 16 bit word to the selected hardware control port.

```
code pl@        \ port -- x ; read port
```
Read 32 bits from the hardware control port supplied.

```
code pl!        \ x port -- ; write port
```
Write the supplied 32 bits to the selected hardware control port.

```
: +Ports        \ port #ports -- ior
```
Enable access to a range of ports starting at *port*. Return 0 on success. Port numbers must be in the range 0..$3FF. You must have root permissions.

```
: -Ports        \ port #ports -- ior
```
Disable access to a range of ports starting at *port*. Return 0 on success. Port numbers must be in the range 0..$3FF. You must have root permissions.

```
: PlayNote      \ hertz ms --
```
Play a note on the internal PC speaker. Ports $42, $43 and $61 must be enabled first.

`: pio-test      \ --`

A test routine for hardware access. Enables ports $40..$6F and confirms access. If you hear a familiar tune, all is well!

11.13 Program launch status

Programs can be launched in several ways.

- From a shell in foreground mode: it has a terminal.
- From a shell or script in background mode, e.g. `vfxlin &`: it shares a terminal which is normally only useful for output.
- From the init process or detached by another process. There is no terminal at all.

This code allows you to determine how the program was launched.

`0 value AppPPID         \ -- x`

This application's parent's process ID.

`0 value AppPGRP         \ -- x`

This application's process group.

`0 value ctPGRP          \ -- x`

The controlling terminal's process group.

`0 value AppLaunch       \ -- x`

How the application was launched:

- -1 - detached
- 0 - backgrounded
- 1 - foreground

`: TestLaunch    \ --`

Set the data above to determine how the application was launched. Run at program launch.

11.14 Folders and Files

`: dirExists?    \ caddr len -- flag`

Return true if a directory exists. Macros are expanded.

`: create-dir    \ caddr len -- ior`

Create a directory, returning zero on success. Macros are expanded. Default permissions are used.

`: forceDir      \ caddr len -- ior`

Create the directory if it does not exist. Macros are expanded.

`: copy-file     \ src srclen dest destlen nooverwrite -- ior`

Copy the file. If *nooverwrite* is non-zero and the destination exists, an error is returned.

12 Intel 386+ Assembler

VFX Forth has a built-in assembler. This is to enable you to write time-critical definitions - if time is a constraint - or to do things that might perhaps be more difficult in Forth - things such as interrupt service routines. The assembler supports the 80386 and the 80387 chips. Definitions written in assembler may use all the variables, constants, etc. used by the Forth system, and may be called from the keyboard or from other words just like any Forth high-level word. It is important when writing a code definition to remember which machine registers are used by the Forth system itself. These registers are documented later in this chapter. All other registers may be used freely. The reserved registers may also be used - but their contents must be preserved while they are in use and reset afterwards.

The assembler mnemonics used in the Forth assembler are just the same as those documented in the Intel literature. The operand order is also the same. The only difference is the need for a space between each portion of the instruction. This is a requirement of the Forth interpreter.

The assembler has certain defaults. These cover the order of the operands, the default addressing modes and the segment size. These are described later in this chapter.

12.1 Using the assembler

Normally the assembler will be used to create new Forth words written in assembler. Such words use `CODE` and `END-CODE` in place of `:` and `;` or `CREATE` and `;CODE` in place of `CREATE` and `DOES>`.

The word `CODE` creates a new dictionary header and enables the assembler.

As an example, study the definition of `0<` in assembly language. The word `0<` takes one operand from the stack and returns a true value, -1, if the operand was less than zero, or a false value, 0, if the operand was greater than or equal to zero.

```
CODE 0< \ n - t/f ; define the word 0<
  OR EBX, EBX   \ use OR to set flags
  L,            \ less than zero ?
  IF,           \ y:
    MOV EBX, # -1       \   -1 is true flag
  ELSE,         \ n:
    SUB EBX, EBX        \  dirty set to 0
  ENDIF,
  NEXT,         \ return to Forth
END-CODE
```

Notice how the word `NEXT,` is used. `NEXT,` is a macro that assembles a return to the Forth inner interpreter. All code words must end with a return to the inner interpreter. The example also demonstrates the use of structuring words within the assembler. These words are pre-defined macros which implement the necessary branching instructions. The next example shows the same word, but implemented using local labels instead of assembler structures for the control structures.

```
CODE 0<          \ n - t/f ; define the word 0<
  OR EBX, EBX    \ use OR to set flags
  JGE L$1        \ skip if AX>=0
  MOV EBX, # -1         \ -1 is true flag
  JMP L$2        \ this part done
L$1:             \ do following otherwise
  SUB EBX, EBX   \ dirty set to 0
L$2:
  NEXT,          \ return to Forth
END-CODE
```

12.2 Assembler extension words

There are several useful words provided within VFX Forth to control the use of the assembler.

`;code       \ --`

Used in the form:

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

Stops compilation, and enables the assembler. This word is used with `CREATE` to produce defining words whose run-time portion is written in code, in the same way that `CREATE ... DOES>` is used to create high level defining words.

The data structure is defined between `CREATE` and `;CODE` and the run-time action is defined between `;CODE` and `END-CODE`. The current value of the data stack pointer is saved by `;CODE` for later use by `END-CODE` for error checking.

When `<namex>` executes the address of the data area will be the top item of the CPU call stack. You can get the address of the data area by `POP`ing it into a register.

A definition of `VARIABLE` might be as follows:

```
: VARIABLE
  CREATE 0 ,
;CODE
  sub     ebp, 4
  mov     0 [ebp], ebx
  pop     ebx
  next,
END-CODE

VARIABLE TEST-VAR
```

`CODE       \ --`

A defining word used in the form:

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

Creates a dictionary entry for `<name>` to be defined by a following sequence of assembly language words. Words defined in this way are called **code** definitions. At compile-time, `CODE` saves the data stack pointer for later error checking by `END-CODE`.

```
END-CODE   \ --
```

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

```
LBL:       \ --
```

A defining word that creates an assembler routine that can be called from other code routines as a subroutine. Use in the form:

```
LBL: <name>
  ...code...
END-CODE
```

When `<name>` executes it returns the address of the first byte of executable code. Later on another code definition can call `<name>` or jump to it.

12.3 Dedicated Forth registers

The Forth virtual machine is held within the processor register set. Register usage is as follows:

EAX	scratch
EBX	cached top of data stack
ECX	scratch
EDX	scratch
ESI	Forth User area pointer
EDI	Forth local variable pointer
EBP	Forth data stack pointer - points to NOS
ESP	Forth return stack pointer

All unused registers may be freely used by assembler routines, but they may be altered by the operating system or wrapper calls. Before calling the operating system, all of the Forth registers should be preserved. Before using a register that the Forth system uses, it should be preserved and then restored on exit from the assembler routine. Be aware, in particular, that callbacks will generally modify the EAX register since this is used to hold the value returned from them.

12.4 Default segment size

```
USE32
USE16
```

The first of these specifies that code from that point onwards is for a 32-bit segment. The second directive specifies that, from that point onwards, code generated is for a 16-bit segment. The default is `USE32`. These directives should be used outside a code definition, not within a definition.

It is possible to override the default segment size on an instruction-by-instruction basis. This is detailed later.

12.5 Assembler syntax

12.5.1 Default assembler notation

The assembler is designed to be very closely compatible with MASM and other assemblers. To this end the assembler assembles code written in the conventional prefix notation. However, because code may be converted from other MPE Forth systems, the postfix notation is also supported. The default mode is prefix. The directives to switch mode are as follows:

`PREFIX`

`POSTFIX`

These switch the assembler from then onwards into the new mode. The directives should be used outside a code definition, not within one. Their use within a code definition will lead to unpredictable results.

The assembler syntax follows very closely that of other 80386 assemblers. The major difference being that the VFX Forth assembler needs white space around everything. For example, where in MASM one might define:

```
MOV EAX,10[EBX]
```

we must write:

```
MOV EAX , 10 [EBX]
```

This distinction must be borne in mind when reading the following addressing mode information.

12.5.2 Register to register

Many instructions have a register to register form. Both operands are registers. Such an instruction is of the form:

```
  MOV EAX , EBX
```

This moves the contents of EBX into EAX. For compatibility with older MPE assemblers the first operand may be merged with the comma thus:

```
  MOV EAX, EBX
```

This use of a register name with a 'built-in' comma also applies to other addressing modes.

12.5.3 Immediate mode

If the assembler is set for direct-as-default (the `MPE` directive has been used), immediate numbers must be defined explicitly. This is done by the use of a hash (#) character:

```
MOV EAX , # 23
```

This example places the number 23 in EAX. The directives `OFFSET` and `SEG` are synonyms for `#`.

By default, the assembler is set to immediate-as-default (the `INTEL` directive has been used). In this case immediate numbers do not have to be specifically defined:

```
MOV EAX , 23
```

The above code also places the number 23 in EAX.

12.5.4 Direct mode

If the assembler is set for direct-as-default (the `MPE` directive has been used), direct addresses need not be defined explicitly:

```
MOV EAX , 23
```

This example places the contents of address 23 in EAX. If the assembler is set for immediate-as-default (the `INTEL` directive has been used), direct addresses have to be specifically defined, using the `PTR` or `[]` directives:

```
MOV EAX , PTR 23
MOV EAX , [] 23
```

Both the above code fragments also place the contents of address 23 in EAX.

12.5.5 Base + displacement

Intel define an addressing mode using a base and a displacement. In this mode, the effective address is calculated by adding the displacement to the contents of the base register. An example:

```
MOV EBX , # 0100
MOV EAX , 10 [EBX]
```

In this example, EAX is filled with the contents of address 0100+10, or address 110.

The assembler lays down different modes for displacements of 8-bit or 32-bit size, but this is internal to the assembler. The following registers may be used as base registers with a displacement:

`[EAX] [ECX] [EDX] [EBX] [EBP] [ESI] [EDI]`

If the displacement is zero then the assembler internally defines the mode as Base only. However, the displacement of zero must be supplied to the assembler:

```
MOV EBX , # 0100
MOV EAX , 0 [EBX]
```

This places in EAX the contents of address 100 (pointed to by EBX).

The following registers may be used as a base with no displacement:

`[EAX] [ECX] [EDX] [EBX] [ESI] [EDI]`

12.5.6 Base + index + displacement

The 80386 also allows two registers to be used to indirectly address memory. These are known as the base and the index. Such instructions are of the form:

```
MOV EAX , # 100
MOV EBX , # 200
MOV EDX , 10 [EAX] [EBX]
```

This will place in EDX the contents of address 100+200+10, or address 310. EAX is the base and EBX is the index. Again, the displacement may be 8-bits, 32-bits or have a value of zero. The assembler distinguishes between these three cases. The base and index registers may be any of the following:

```
[EAX] [EBX] [ECX] [EDX] [ESI] [EDI]
```

In addition, [EBP] may be used as the index register, and [ESP] may be used as the base register.

12.5.7 Base + index*scale + displacement

The 80386 further supports an addressing mode where the index register is automatically scaled by a fixed amount - either 2, 4 or 8. This is designed for indexing into two-dimensional arrays of elements of size greater than byte-size. One register may be used as the first index, another for the second index, and the word size becomes implicit in the instruction. The form of this addressing mode is very similar to that outlined above, with the exception that the index operand includes the number which is the scale:

```
MOV EBX , # 100
MOV ECX , # 2
MOV EAX , 10 [EBX] [ECX*4]
```

This stores into EAX, the contents of address 100+(4*2)+10, or address 118. The list of registers which may be used as base is the same as the above. The list of scaled indexes is as follows:

```
[EAX*2] [ECX*2] [EDX*2] [EBX*2] [EBP*2] [ESI*2] [EDI*2]
[EAX*4] [ECX*4] [EDX*4] [EBX*4] [EBP*4] [ESI*4] [EDI*4]
[EAX*8] [ECX*8] [EDX*8] [EBX*8] [EBP*8] [ESI*8] [EDI*8]
```

12.5.8 Segment overrides

Some instructions may be prefixed with a segment override. These force data addresses to refer to a segment other than the data segment. The override must precede the instruction to which it relates:

```
MOV EBX , # 100
ES: MOV EAX , 10 [EBX]
```

This will set EAX to the value contained in address 110 in the extra segment. The list of segment overrides is:

```
CS: DS: ES: FS: GS: SS:
```

12.5.9 Data size overrides

The default data size for a `USE32` segment is 32-bit, but the default data size for a `USE16` segment is 16-bit. These are the default data sizes the assembler will use. If the data is of a different size a data size override will have to be used. To define the size of the data the following size specifiers are used:

```
BYTE or B.
WORD or W.
DWORD or D.
QWORD
TBYTE
FLOAT
DOUBLE
EXTENDED
```

It is only necessary to specify size when ambiguity would otherwise arise. For example:

```
MOV   0 [EDX], # 10  \ can't tell
MOV   0 [EDX], EAX   \ EAX specifies
```

The `BYTE` size defines that a byte operation is required:

```
MOVZX EAX , BYTE 10 [EBX]
```

The abbreviation `B.` may also be used in place of `BYTE` to define a byte operation. The `WORD` specifier defines that 16-bits are required:

```
MOV AX , WORD 10 [EBX]
```

The abbreviation `W.` may also be used to define a word operation. `DWORD` is the default for a `USE32` segment, and indicates that 32-bit data is to be used:

```
MOV EAX , DWORD 10 [EBX]
FSTP DWORD 10 [EBX]
```

The abbreviation `D.` may also be used to specify a `DWORD` operation. The remaining size specifiers define data sizes for the floating point unit.

`QWORD` defines a 64-bit operation:

```
FSTP QWORD 10 [EBX]
```

TBYTE defines a 10-byte (80-bit) operation, such as:

```
FSTP TBYTE 10 [EBX]
```

`FLOAT`, `DOUBLE` and `EXTENDED` are synonyms for `DWORD`, `QWORD` and `TBYTE` respectively.

The segment type defines the default data size and address size for the code in the segment. If needed, it is possible to force the data size or the address size laid down to be the other. There is a set of data and address size overrides which work for one instruction only. These are:

```
D16:
D32:
A16:
A32:
```

and they would be used as follows:

```
D16: MOV EAX , # 23
A16: MOV EAX , 10 [EBX]
```

The first of these, in a `USE32` segment, would lay down 16-bit data to be loaded into AX. The second would lay down a 16-bit offset from [EBX] for the effective address in the instruction. The situation would be reversed in a `USE16` segment - the `A32:` and `D32:` directives would cause 32-bit data or addresses to be laid.

12.5.10 Near and far, long and short

Jumps and branches may be either intra-segment or inter-segment. The former is a short branch or call whilst the latter is a long branch or call. The assembler is able to lay down either form. The default for a JMP or a CALL is near, whilst the default for a conditional branch is short. RET follows the same pattern as CALL. The directives supporting short/long and near/far are:

```
SHORT  LONG  NEAR  FAR
```

These would be used as follows:

```
2 CONSTANT THAT         \ the segment number
LBL: THIS               \ the address

CALL THIS
CALL NEAR THIS
CALL FAR THAT THIS
JMP THIS
JMP NEAR THIS
JMP FAR THAT THIS

JCC THIS
JCC SHORT THIS
JCC LONG THIS

RET THIS
RET NEAR THIS
RET FAR THAT THIS
```

For compatibility with older MPE assemblers the mnemonics `CALL/F`, `RET/F` and `JMP/F` are also provided.

12.5.11 Syntax exceptions

The assembler in VFX Forth follows both the syntax and the mnemonics defined in the Intel Programmers Reference books, for both the 80386 and the 80387. However, there are certain exceptions. These are listed below.

The zero operand forms of certain stack register instructions for the 80387 have been omitted. Their functionality is supported however. Such instructions are listed below, with a form of the syntax which will support the function:

```
FADD    FADDP ST(1) , ST
FCOM    FCOM ST(1)
FCOMP   FCOMP ST(1)
FDIV    FDIVP ST(1) , ST
FDIVR   FDIVRP ST(1) , ST
FMUL    FMULP ST(1) , ST
FSUB    FSUBP ST(1) , ST
FSUBR   FSUBRP ST(1) , ST
Certain 80386 instructions have either one operand or two operands, of which
only one is variable. These instructions are:
```

```
MUL DIV IDIV NEG NOT
```

These instructions take only one operand in the VFX Forth assembler.

12.5.12 Local labels

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

Ten labels are pre-defined, and their names are fixed. Additional labels can be defined up to a maximum of 32. There is a limit of 128 forward references. A reference to a label is valid until the next occurrence of `LBL:`, `CODE` or `;CODE`, whereupon all the labels are reset.

A reference to a label in a definition must be satisfied in that definition. You cannot define a label in one code definition and refer to it from another.

The local labels have the names `L$1 L$2 ... L$10` and these names should be used when referring to them e.g.

```
JNE L$5
```

A local label is defined by words of the same names, but with a colon as a suffix:

```
L$1: L$2: ... L$10:
```

Additional labels (up to a maximum of 32 altogether) may be referred to by:

```
n L$
```

where n is in the range 11..32 (decimal), and they may be defined by:

```
n L$:
```

where n is again in the range 11..32 (decimal).

12.5.13 CPU selection

This assembler is designed to cope with CPUs from 80386 upwards. Some instructions are only available on later CPUs. Note that CPU selection affects the assembler and the VFX code code generator, **not** the run time of your application. If you select a higher CPU level than the application runs on, incorrect operation will occur.

```
CPU=386      \ -- ; select base instruction set
CPU=PPro     \ -- ; Pentium Pro and above with CMOVcc
CPU=P4       \ -- ; Pentium 4 and above
PPro?        \ -- flag ; true if at least Pentium Pro
P4?          \ -- flag ; true if at least Pentium 4
```

The VFX code generator also uses this information to enable various code generation techniques. For VFX Forth for DOS, the default selection is for 386 class CPUs, for all others it is for the Pentium 4 instruction set.

12.6 Assembler structures

Structures like the Forth control structures have been added to the assembler. They allow forward branches without the need for labels and impose the strictures of structured programming to the assembler level. Devotees of spaghetti programming are free to go their own way as the copious supply of branch instructions are still available, and the local label facility may be used with all branch instructions.

The status flag indicator required must prefix conditional structures. The structure assembled will have a branch opcode that is the logical inverse of the one specified. Thus for `EQ, IF,` a JNE will be assembled so that the code after `IF,` is executed if the EQ status occurs. The assembler structure words end in a comma e.g. `IF,` to differentiate them from the regular Forth structures, and to indicate that code is being generated.

The structures are described below, and the symbol cc condition code) may be any one of the following:

`Z,` equal to 0

`NZ,` not equal to 0

`S,` less than 0

`NS,` greater than or equal to 0

`L,` less than

`GE,` greater than or equal

`LE,` less than or equal

`G,` greater than

`B,` unsigned less than - address compares

`AE,` unsigned greater than or equal

`BE,` unsigned less than or equal

`A,` unsigned greater than

`O,` overflow

`NO,` no overflow

`PE,` parity even

`PO,` parity odd

`CY,` carry flag set

`NC,` carry flag not set

`NCXZ,` ECX/CX register non-zero

The structure words build sets of assembler branches to perform functions equivalent to their high-level namesakes, but the names end with a comma to distinguish them from the high-level Forth words. Be sure you understand the high level structures before using the assembler equivalents. The structures are:

```
cc IF, ... THEN,
cc IF, ... ELSE, ... THEN,

BEGIN, ... AGAIN,
BEGIN, ... cc UNTIL,
BEGIN, ... cc WHILE, ... REPEAT,
```

An additional structure allows a section of code to be performed 'n' times. All it actually does is to load ECX with 'n'and mark the start of a backward branch so that the mark may be used later. The structure is:

```
n TIMES, ... LOOP,
```

12.7 Assembler mode switches

`: mpe             \ -- ; force def # addressing`

Select the MPE default addressing mode, in which the default addressing mode is direct addressing. This is provided for compatibility with legacy MPE systems. The indicators `[]` and `#` can be used for code which must be compiled in either condition.

`: intel           \ -- ; force def.direct addressing`

Select the INTEL default addressing mode, in which the default addressing mode is immediate addressing. The indicators `[]` and `#` can be used for code which must be compiled in either condition.

`: prefix          \ -- ; select prefix mode`

Set the assembler to use prefix notation with the opcode first. This is the default condition.

`: postfix         \ -- ; select postfix mode`

Set the assembler to use postfix notation with the opcode after the operands.

12.8 Macros and Assembler access

Because of the performance of the VFX optimiser, use of assembler is only necessary when defining new compilation structures. Otherwise the use of assembler code should be avoided.

Assembler macros are defined as follows:

```
MASM: <name>
  <assembler code goes here>
;MASM
e.g. the following macro pops the top of the NDP stack to the external
floating point stack.
MASM: popFPU    \ -- ; pops FTOS to float stack
  mov   eax, FSP-OFFSET [esi]   \ get FP stack pointer
  lea   eax, -FPCELL [eax]      \ update stack pointer
  fstp  fword 0 [eax]           \ store and pop
  mov   FSP-OFFSET [esi], eax   \ restore FP stack pointer
;MASM
```

In line assembler code may be compiled into the middle of a colon definition by using the phrase:

```
: <name>      \ just another Forth word
  ...
  [ASM  <insert assembler here>  ASM]
  ...
;
```

A fragment of assembler for compilation when the containing word is executed can be defined by using the following:

```
: a-compiler  \ will compile some assembler
  ...
  a[ <fragment to be compiled> ]a
  ...
;
```

The following example compiles an in-line floating point literal.

```
: o_flit,       \ F: f -- ; F: -- f ; compile floating point literal
  a[ popFPU                        \ references a previous macro
     jmp   here 2+  FPCELL +       \ skip inline literal
  ]a
  f,
  a[ fld   fword ptr here FPCELL - ]a
;
```

When using in-line code generators such as `[ASM ...ASM]` you should flush the code generator contents with `[O/F]`.

```
[O/F] [ASM ... ASM]
```

After [ASM the top of the data stack will be in EBX with all other stack items pointed to by EBP. The code generator expects this same state to exist after ASM].

```
: dxb           \ b -- ; lay byte
```
Lay a byte into the instruction stream. Use in the form:

```
dxb $55
```

```
: dxw           \ w -- ; lay 16 bits
```
Lay a 16-bit word into the instruction stream. Use in the form:

```
dxw $55AA
```

```
: dxl           \ l -- ; lay 32 bit long
```
Lay a 32-bit dword into the instruction stream. Use in the form:

```
dxl $11223344
```

```
: $            \ -- chere
```
Return the PC value of the start of the instruction.

12.9 Assembler error codes

```
#-701 Invalid addressing mode
#-702 N not in range -128..+127
#-703 Label reference number out of range
#-704 Label definition number out of range
#-705 Invalid instruction for selected CPU type
```

13 Intel 386+ Disassembler

The VFX Forth system includes a disassembler for debugging purposes. any native code built by the system can be viewed at a machine code level.

```
: al-init-dis   \ addr len -- ; initialise disassembly
```
Initialise the disassembly range before using (DASM).

```
: ft-init-dis   \ from to -- ; initialise disassembly
```
Initialise the disassembly range before using (DASM).

```
: 1DISASM       \ --
```
Disassemble the next instruction. The range has already been set.

```
: (dasm)        \ --
```
Disassemble a block of code whose range has already been set.

13.1 Low-Level Disassembly Words

```
: disasm/f      \ addr --
```
Disassemble memory starting at ADDR.

```
: disasm/ft     \ from to --
```
Disassemble memory between the memory addresses FROM and TO.

```
: DISASM/al     \ addr len --
```
Disassemble LEN bytes of code starting at memory address ADDR.

```
: dasm          \ -- ; DASM <word>
```
Disassemble a given definition

13.2 Higher Level Disassembly

VFX Forth also contains some higher-level words to aid disassembly. These words attempt to interpret the raw assembler code as provided by `DASM` to identify items such as inline strings and `USER` variables.

```
: dis           \ -- ; DIS <name>
```
Disassemble a supplied target word using the higher level interpreter to aid readability.

```
: see           \ -- ; SEE <name>
```
An alias for `DIS` supplied for compatibility with other Forth systems.

14 Floating Point

14.1 Introduction

The floating point packages use the FPU instruction set. The source code can be found in the files *Lib\Ndp387.fth*, *Lib\Hfp387.fth* and *Lib\HfpGL32.fth*. *Ndp387.fth* is the primary package, and produces the fastest and smallest code.

As of v4.43, the floating point output routines have been rewritten in *Lib\Ndp387.fth and *\i{Lib\Hfp387.fth*. The previous code used up to v4.42 is retained in the files *Lib/Ndp387.v442.fth* and *Lib/Hfp387.v442.fth*.

Additional support for floating point handling plus a significant library of numeric code may be found in the Forth Scientific Library. Look in the folder *Lib\FSL*.

14.1.1 Ndp387.fth - coprocessor stack

In *Ndp387.fth* floating point numbers are kept in the floating point unit's internal stack only. This code is significantly faster than when using an external stack, but is limited to the use of 8 floats on the NDP stack, including any working temporary numbers.

By default, *Ndp387.fth* defines floating point stack items and literals to be in 80 bit extended real format. If you need to save (a small amount of) space, the default can be changed by setting the constant `FPCELL` to 8 for 64 bit double precision or to 4 for 32 bit single precision. If you do this, accuracy and resolution may/will suffer.

From VFX Forth 3.70 onwards, *Ndp387.fth* includes an optimiser. According to the results of *Examples\Benchmrk\mm.fth* this nearly doubles the overall performance of the matrix multiply floating point benchmark code. Well tuned algorithms may see speed improvements of over three times. Use of the Pentium optimisations with `+IDATA` is recommended for performance critical floating point code.

The file *Ndp387.v36.fth* contains the last release of the unoptimised code.

14.1.2 Hfp387.fth - external FP stack

In *Hfp387.fth* floating point numbers are kept on a separate stack, pointed to by the `USER` variables `FSP` and `FS0`. The top of the FP stack is cached in the FPU.

All tasks, winprocs and callbacks are allocated a separate 4096 byte floating point stack. If you need a larger one, allocate it from the heap using `ALLOCATE`, and modify `FSP` and `FS0` accordingly. Note that the stack grows down.

By default, *Hfp387.fth* defines floating point stack items and literals to be in 80 bit extended real format. If you need to save a bit of space, the default can be changed by setting the constant `FPCELL` to 8 for 64 bit double precision or to 4 for 32 bit single precision. If you do this, accuracy and resolution may/will suffer.

If you are uncertain of the state of the floating point unit, you can initialise it using `FINIT ( -- )`. After `FINIT`, the internal floating point stack is empty.

14.1.3 HfpGL32.fth - 32 bit floats on data stack

This file is specifically designed for interfacing to some OpenGL APIs. Floating point numbers are kept as 32 bit floats on the data stack. Compared to the previous two packages, *HfpGL32.fth* has a restricted range of facilities and is a subset of the other two packages. See the source code for more details.

14.2 Radians and Degrees

Please note that the trig functions are calculated in radians, for calculations in degrees use `DEG>RAD` beforehand, `RAD>DEG` afterwards.

14.3 Number formats, ANS and Forth200x

The ANS Forth standard specifies that floating point numbers must be entered in the form 1.234e5 and must contain a point '.' and 'e' or 'E', and that double integers are terminated by a point '.'.

This situation prevents the use of the standard conversion words in international applications because of the interchangable use of the '.' and ',' characters in numbers. Because of this, VFX Forth uses two four-byte arrays, `FP-CHAR` and `DP-CHAR`, to hold the characters used as the floating point and double integer indicator characters. By default, `FP-CHAR` is initialised to '.' and `DP-CHAR` is initialised to to ',' and '.'. For strict ANS compliance, you should set them as follows:

```
\ ANS standard setting
  char . dp-char !
  char . fp-char !
: ans-floats    \ -- ; for strict ANS compliance
  [char] . dp-char !
  [char] . fp-char !
;
\ MPE defaults
  char , dp-char !
  char . dp-char 1+ c!
  char . fp-char !
: mpe-floats    \ -- ; for existing and most legacy code
  [char] , dp-char !
  [char] . dp-char 1+ c!
  [char] . fp-char !
;
\ Legacy defaults, including ProForth
  char , dp-char !
  char . fp-char !
: legacy-floats \ -- ; for legacy code
  [char] , dp-char !
  [char] . fp-char !
;
```

You can of course set these arrays to hold any values which suit your application's language and locale. Note that integer conversion is always attempted before floating point conversion. This means that if the FP-CHAR and DP-CHAR arrays contain the same character, floating point numbers must contain 'e' or 'E'. If the arrays are all different, a number containing the FP-CHAR will be successfully converted as a floating point number, even if it does not contain 'e' or 'E'.

As of January 2007, recommendations made to the Forth200x standards effort have been adopted by MPE for REPRESENT. The impact of these changes is that the minimum buffer size used for REPRESENT should be at least #FDIGITS characters, normally 18 bytes. For details of the proposal, see:

```
Examples/usenet/Ed/Represent_20.txt
Examples/usenet/Ed/Represent_30.txt
```

14.4 Floating point exceptions

Exception handling is determined by the operating system. On current Windows platforms, floating point exceptions are not generated by the NDP. This can be changed by altering the bottom 6 bits of the NDP control word using CW@ and CW!.

By default, the system prompt will report exception status, and clear the pending exception status. Exception status reporting does not mean that the Windows exception handler has been triggered, it only means that the status flag has been set.

14.5 Standards compliance, F>S and F>D

After much discussion on the comp.lang.forth newsgroup, a consensus was reached that F>D and F>S must truncate to zero. This is also the behaviour required by the Forth Scientific Library (FSL). Historically, MPE floating point packs permit the integer rounding mode to be set by the user. In order to support both camps, VFX Forth now behaves as follows:

- F>D and F>S truncate to zero,
- FR>D and FR>S follow the current rounding mode.

14.6 Configuration

```
0 value FpSystem
```

The value FPSYSTEM defines which floating point pack is installed and active. Each floating point pack defines its own type as follows:

- 0 constant NoFPSystem
- 1 constant HFP387System
- 2 constant NDP387System
- 3 constant OpenGL32System

```
#10 constant FPCELL     \ -- n
```

Defines the size of literals and floating point numbers in memory and on floating point stacks in memory. FPCELL can be changed to 8 for 64 bit double precision or to 4 for 32 bit single precision. If you do this, accuracy and resolution may/will suffer.

```
  constant #fdigits     \ -- u
```

Returns the largest number of usable digits available from REPRESENT. Equivalent to the environment variable MAX-FLOAT-DIGITS.

```
false constant [fpdebug] immediate
```

Set this true when compiling *NDP387.FTH*, and a debug build will be constructed. In this state, the state of the FPU is checked after each word. If a floating point exception has been generated, a diagnostic is issued, and the system aborts. Set this only when testing. Note that the NDP387 optimiser may well cause this to be ignored.

```
  defer fpcheck                          \ see later for real action
```
A DEFERred word called at the end of `CODE` routines when `[FPDEBUG]` is non-zero.

```
variable signed-zero    \ --addr
```
Set non-zero to display signed-zero.

14.7 Assembler macros

```
: fword         \ --
```
Selects appropriate floating point size for the assembler. Note that this is defined by the constant `FPCELL`. `FWORD` will be a synonym for `FLOAT`, `DOUBLE` or `TBYTE`.

```
: fnext,        \ -- ; can be changed for debugging
```
The equivalent of `NEXT,` for floating point routines. If `[FPDEBUG]` is non-zero, a call to `FPCHECK` is assembled.

14.8 Optimiser support

This code is only provided for *Ndp387.fth* in VFX Forth 3.70 onwards.

```
0 value FPsin?  \ -- flag
```
Returns non-zero if source inlining is permitted for words containing floating point code sequences. By default, FP source inlining is disabled.

```
: [+FPsin       \ -- x
```
Start a `[+FPSIN ... FPSIN]` section in which new FP code can be source inlined.

```
: [-FPsin       \ -- x
```
Start a `[-FPSIN ... FPSIN]` section in which new FP code cannot be source inlined.

```
: FPsin]        \ x --
```
End a `[+/-FPSIN ... FPSIN]` section. The previous FP source inliner state is preserved.

```
: fseq:         \ -- ; FSEQ: <name> ... ;FSEQ
```
Start an assembler sequence which is compiled for `<name>`.

```
: ;fseq         \ -- ; FSEQ: <name> ... ;FSEQ
```
Ends an assembler sequence started by `FSEQ:`.

14.9 FP constants

```
code %0         \ F: -- f#(0)
```
Floating point 0.0

```
code %1         \ F: -- f#(1)
```
Floating point 1.0

```
code %pi        \ F: -- f#(pi)
```
Floating point PI

```
code %pi/2      \ F: -- f#(pi/2)
```
Floating point PI/2

```
code %pi/4      \ F: -- f#(pi/4)
```

Floating point PI/4

`code %lg2e      \ F: -- log2(e)`
Returns log (base 2) of e.

14.10 FP control operations

`code finit      \ F: -- ; resets FPU`
Reset the floating point unit and NDP stack.

`code cw@        \ -- cw ; get NDP control word`
Return the floating point unit Control Word.

`code cw!        \ cw -- ; set NDP control word`
Set the floating point unit Control Word.

`code sw@        \ -- sw ; get NDP status word`
Return the floating point unit Status Word.

`code fclex      \ -- ; clear exceptions`
Clear any pending floating point exceptions.

14.11 FP Stack operations

`code fdup       \ F: f -- f f`
Floating point equivalent of `DUP`.

`code fswap      \ F: f1 f2 -- f2 f1`
Floating point equivalent of `SWAP`.

`code F2SWAP     \ F: r1 r2 r3 r4 -- r3 r4 r1 r2`
Floating point equivalent of `2SWAP`.

`code fdrop      \ F: f --`
Floating point equivalent of `DROP`.

`code fover      \ F: f1 f2 -- f1 f2 f1`
Floating point equivalent of `OVER`.

`code frot       \ F: f1 f2 f3 -- f2 f3 f1`
Floating point equivalent of `ROT`.

`code fpick      \ n -- ; F: -- f`
Floating point equivalent of `PICK`. Note that because the pick index is an integer, it is on the normal Forth integer data stack, and the result, being a floating point number, is on the floating point stack.

`code ndepth     \ -- n ; depth of NDP stack`
Returns on the Forth data stack the number of items on the FPUs's internal working stack.

`: fdepth        \ -- #f`
Floating point equivalent of `DEPTH`. The result is reurned on the Forth data stack, **not** the float stack.

14.12 Memory operations SF@ SF! DF@ DF! etc

`code f@         \ addr -- ; F: -- f`
Places the contents of addr on the float stack. The size of the item fetched was defined by `FPCELL` at compile time.

```
code sf@        \ addr -- ; F: -- f
```
Places the 32 bit float at addr on the float stack.

```
code df@        \ addr -- ; F: -- f
```
Places the 64 bit double float at addr on the float stack.

```
code tf@        \ addr -- ; F: -- f
```
Places the 80 bit extended float at addr on the float stack.

```
code f!         \ addr -- ; F: f --
```
Stores the top of the float stack as an `FPCELL` sized number at addr.

```
code sf!        \ addr -- ; F: f --
```
Stores the top of the float stack as an 32 bit float number at addr.

```
code df!        \ addr -- ; F: f --
```
Stores the top of the float stack as an 64 bit double float number at addr.

```
code tf!        \ addr -- ; F: f --
```
Stores the top of the float stack as an 80 bit extended float number at addr.

```
code f+!        \ F: f -- ; addr -- ; add f to data at addr
```
Add F to the data at ADDR.

```
code f-!        \ F: f -- ; addr -- ; sub f from data at addr
```
Subtract F from the data at ADDR.

```
code sf+!       \ F: f -- ; addr -- ; add f to data at addr
```
Add F to the 32 bit float at ADDR. NDP387.FTH only.

```
code sf-!       \ F: f -- ; addr -- ; sub f from data at addr
```
Subtract F from the 32 bit float at ADDR. NDP387.FTH only.

```
code df+!       \ F: f -- ; addr -- ; add f to data at addr
```
Add F to the 64 bit float at ADDR. NDP387.FTH only.

```
code df-!       \ F: f -- ; addr -- ; sub f from data at addr
```
Subtract F from the 64 bit float at ADDR. NDP387.FTH only.

```
code tf+!       \ F: f -- ; addr -- ; add f to data at addr
```
Add F to the 80 bit float at ADDR. NDP387.FTH only.

```
code tf-!       \ F: f -- ; addr -- ; sub f from data at addr
```
Subtract F from the 80 bit float at ADDR. NDP387.FTH only.

```
code f@+        \ addr -- addr' ; F: -- f
```
Places the contents of addr on the float stack and increments the address. The size of the item fetched and the increment is defined by `FPCELL`. NDP387.FTH only.

```
code sf@+       \ addr -- addr' ; F: -- f
```
Places the 32 bit float at addr on the float stack, and increments addr by 4. NDP387.FTH only.

```
code df@+       \ addr -- addr' ; F: -- f
```
Places the 64 bit float at addr on the float stack, and increments addr by 8. NDP387.FTH only.

```
code tf@+       \ addr -- addr' ; F: -- f
```
Places the 80 bit float at addr on the float stack, and increments addr by 10. NDP387.FTH only.

```
code f!+        \ addr -- addr' ; F: f --
```

Stores the top of the float stack as an `FPCELL` sized number at addr, and updates addr appropriately. NDP387.FTH only.

`code sf!+        \ addr -- addr' ; F: f --`

Stores the top of the float stack as a 32 bit float at addr, and updates addr appropriately. NDP387.FTH only.

`code df!+        \ addr -- addr' ; F: f --`

Stores the top of the float stack as a 64 bit float at addr, and updates addr appropriately. NDP387.FTH only.

`code tf!+        \ addr -- addr' ; F: f --`

Stores the top of the float stack as an 80 bit float at addr, and updates addr appropriately. NDP387.FTH only.

14.13 Dictionary operations

`: tf,            \ F: f --`

Lays an 80 bit extended float into the dictionary, reserving 10 bytes

`: df,            \ F: f --`

Lays an 64 bit double float into the dictionary, reserving 8 bytes

`: sf,            \ F: f --`

Lays a 32 bit float into the dictionary, reserving 4 bytes

`: f,             \ F: f --`

lays a default float into the dictionary, reserving `FPCELL` bytes

`: falign         \ --`

Aligns the dictionary to accept a default float.

`: faligned       \ addr -- addr'`

Aligns the address to accept a default float.

`: float+         \ addr -- addr'`

Increments addr by `FPCELL`, the size of a default float.

`: floats         \ n1 -- n2`

Returns n2, the size of n1 default floats.

`: sfalign        \ --`

Aligns the dictionary to accept a 32 bit float.

`: sfaligned      \ addr -- addr'`

Aligns the address to accept a 32 bit float.

`: sfloat+        \ addr -- addr'`

Increments addr by the size of a 32 bit float.

`: sfloats        \ n1 -- n2`

Returns n2, the size of n1 32 bit floats.

`: dfalign        \ --`

Aligns the dictionary to accept a 64 bit double float.

`: dfaligned      \ addr -- addr'`

Aligns the address to accept a 64 bit float.

`: dfloat+        \ addr -- addr'`

Increments addr by the size of a 64 bit double float.

```
: dfloats       \ n1 -- n2
```
Returns n2, the size of n1 64 bit double floats.

```
: tfalign       \ --
```
Aligns the dictionary to accept an 80 bit extended float.

```
: tfaligned     \ addr -- addr'
```
Aligns the address to accept an 80 bit extended float.

```
: tfloat+       \ addr -- addr'
```
Increments addr by the size of an 80 bit extended float.

```
: tfloats       \ n1 -- n2
```
Returns n2, the size of n1 80 bit extended floats.

14.14 FP defining words

```
: fvariable     \ F: -- ; -- addr
```
Use in the form: `FVARIABLE <name>` to create a variable that will hold a default floating point number.

```
: farray                \ n -- ; i -- addr
```
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.

```
: fconstant     \ F: f -- ; F: -- f
```
Use in the form: `<float> FCONSTANT <name>` to create a constant that will return a floating point number.

```
: fvalue        \ F: f -- ; ??? -- ???
```
Use in the form: `<float> FVALUE <name>` to create a floating point version of `VALUE` that will return a floating point number by default, and that can accept the operators `TO`, `ADDR`, `ADD`, `SUB`, and `SIZEOF`.)

14.15 Basic functions + - * / and others

```
code f+         \ f1 f2 -- f1+f2
```
Floating point add.

```
code f-         \ f1 f2 -- f1-f2
```
Floating point subtract.

```
code f*         \ f1 f2 -- f1*f2
```
Floating point multiply.

```
code f/         \ f1 f2 -- f1/f2
```
Floating point divide.

```
code fmod       \ F: f1 f2 -- f3
```
Floating point modulus. Returns f3 the remainder after repeatedly subtracting f2 from f1. Often used to force arguments to lie in the range: 0 <= arg < f2

```
code fsqrt      \ F: f -- sqrt(f)
```
Floating point square root.

```
code 1/f        \ F: f -- 1/f
```

Floating point reciprocal.

`code fabs        \ F: f -- |f|`
Floating point absolute.

`code fnegate     \ F: f -- -f`
Floating point negate.

`code f2*         \ F: f -- f*2`
Floating point multiply by two.

`code f2/         \ F: f -- f/2`
Floating point divide by two.

14.16 Integer to FP conversion

`code s>f         \ n -- ; F: -- f`
Converts a single integer to a float.

`code d>f         \ d -- ; F: -- f`
Converts a double integer to a float.

`code f>s         \ F: f -- ; -- n ; convert float to integer`
Converts a float to a single integer. Note that `F>S` truncates the number towards zero according to the ANS specification. See `FR>S` below.

`code f>d         \ F: f -- ; -- d ; convert float to double integer`
Converts a float to a double integer. Note that `F>D` truncates the number towards zero according to the ANS specification. See `FR>D` below.

`code fr>s        \ F: f -- ; -- n ; convert float to integer`
Converts a float to a single integer using the current rounding mode.

`code fr>d        \ F: f -- ; -- d ; convert float to double integer`
Converts a float to a double integer using the current rounding mode.

14.17 FP comparisons

`: f0<            \ F: f1 -- ; -- t/f ; less than zero?`
Floating point `0<`. N.B. result is on the Forth integer data stack.

`: f0=            \ F: f1 -- ; -- t/f ; equal zero?`
Floating point `0=`. N.B. result is on the Forth integer data stack.

`: f0<>           \ F: f1 -- ; -- t/f ; not equal zero?`
Floating point `0<>`. N.B. result is on the Forth integer data stack.

`: f0>            \ F: f1 -- ; -- t/f ; greater than zero?`
Floating point `0>`. N.B. result is on the Forth integer data stack.

`: f<             \ F: f1 f2 -- ; -- t/f ; one less than the other?`
Floating point `<`. N.B. result is on the Forth integer data stack.

`: f=             \ F: f1 f2 -- ; -- t/f ; equal each other?`
Floating point `=`. N.B. result is on the Forth integer data stack.

`: f<>            \ F: f1 f2 -- ; -- t/f ; one not equal the other?`
Floating point `<>`. N.B. result is on the Forth integer data stack.

`: f>             \ F: f1 f2 -- ; -- t/f ; one less than the other?`
Floating point `>`. N.B. result is on the Forth integer data stack.

```
: f<=            \ F: f1 f2 -- ; -- t/f ; one less or equal the other?
```
Floating point `<=`. N.B. result is on the Forth integer data stack.

```
: f>=            \ F: f1 f2 -- ; -- t/f ; one greater or equal the other?
```
Floating point `>=`. N.B. result is on the Forth integer data stack.

```
code fsignbit    \ F: f -- ; -- sign
```
Return the sign bit of the floating point number. This is not the same as `f0<` for f=+/-0e0.

```
: fsign          \ F: r1 -- ; -- sign
```
Get the sign of floating point *r1*.

```
: f~             \ F: f1 f2 f3 -- ; -- flag
```
Approximation function. If f3 is positive, flag is true if abs[f1-f2] is less than f3. If f3 is zero, flag is true if the f2 is exactly equal to f1. If f3 is negative, flag is true if abs[f1-f2] less than abs[f3*abs[f1+f2]].

14.18 Words dependent on FP compares

```
: ?fnegate       \ F: f1 f2 -- f3
```
Floating point `NEGATE`.

```
: fmax           \ F: f1 f2 -- f3
```
Floating point `MAX`.

```
: fmin           \ F: f1 f2 -- f3
```
Floating point `MIN`.

14.19 FP logs and powers

```
code flog        \ F: f -- log(f)
```
Floating point log base 10.

```
code fln         \ F: f -- ln(f)
```
Floating point log base e.

```
code 2**         \ F: f -- 2^f
```
Floating point: returns 2^F.

```
code fexp        \ F: f -- e^f ; was called FE^X
```
Floating point e^f.

```
: fexpm1         \ F: f -- (e^f)-1 ; 12.6.2.1516
```
Floating point log base (e^f)-1.

```
code flnp1       \ F: f1 -- f2
```
The output *f2* is the natural logarithm of the input plus one. An ambiguous condition exists if *f1* is less than or equal to negative one.

```
code falog       \ F: f -- 10^f ; was called f10^f, new name: ans
```
Floating point anti-log base 10.

```
code (f**)       \ F: f1 f2 -- f1^f2
```
Floating point returns f1 raised to the power f2. No error checking is performed. If floating point execeptions are masked, which is the default condition, the system will return a NaN for f1<0.

```
: f**            \ F: f1 f2 -- f1^f2
```
Floating point: returns f1 raised to the power f2. If f1<=0e0, 0e0 is returned. This behaviour is required by the Forth Scientific Library.

14.20 Rounding

The default rounding configuration is round to nearest.

`: fround        \ F: f1 -- f1'`
Round the number to nearest or even.

`: ftrunc        \ F: f1 -- f1'`
Round the number towards zero, returning an integer result on the FP stack.

`: fint          \ F: f1 -- f1'`
A synonym for `FTRUNC`. `FINT` will be removed in a future release.

`: floor         \ F: f1 -- f1'`
Floored round towards -infinity.

`: roundup       \ F: f1 -- f1'`
Round towards +infinity.

`: rounded       \ -- ; set NDP to round to nearest`
Set NDP to round to nearest for all operations other than `FINT`, `FLOOR` and `ROUNDUP`.

`: floored       \ -- ; set NDP to floor`
Set NDP to round to floor for all operations other than `FROUND`, `FINT` and `ROUNDUP`.

`: roundedup     \ -- ; set NDP to round up`
Set NDP to round up for all operations other than `FROUND`, `FINT` and `FLOOR`.

`: truncated     \ -- ; set NDP to chop to 0`
Set NDP to chop to 0 for all operations other than `FROUND`, `FLOOR` and `ROUNDUP`.

`code flit       \ F: -- f ; takes floating point number inline`
Followed in line by a floating point number (`FPCELL` bytes) returning this number when executed.

`defer fliteral  \ F: f -- ; F: -- f`
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, whereas `(RLITERAL)` below is not.

`: (rliteral)    \ F: f -- ; F: -- f`
Compiles a float as a literal into the current definition. At execution time, a float is returned. This is the default action of `FLITERAL` above.

14.21 FP trigonometry

`code ftan       \ F: f -- tan(f)`
Floating point tangent.

`code fatan      \ F: f -- atan(f)`
Floating point arctangent.

`code fsin       \ F: f -- sin(f)`
Floating point sine.

`code fasin      \ F: f -- asin(f)`
Floating point arcsine.

`code fcos       \ F: f -- cos(f)`
Floating point cosine.

`code facos      \ F: f -- acos(f)`

Floating point arctangent.

```
code fsincos    \ F: f -- sin(f) cos(f)
```
Returns sine and cosine values of *f*.

```
code fatan2     \ F: f1 f2 -- atan(f1/f2)
```
Floating point arctangent with prior division.

```
: deg>rad       \ F: fdeg -- frad
```
Converts a value in degrees to radians.

```
: rad>deg               \ -- ;
```
Converts a value in radians to degrees.

```
code freduce    \ F: f1 -- f2 ; reduce value to range 0..2pi
```
Reduce f1 to be in the range 0 <= f2 < 2PI.

```
: fcosec        \ F: f -- cosec(f)
```
Floating point cosecant.

```
: fsec          \ F: f -- sec(f)
```
Floating point secant.

```
: fcotan        \ f: f -- cot(f)
```
Floating point cotangent.

```
: fsinh         \ F: f -- sinh(f) ; (e^x - 1/e^x)/2
```
Floating point hyberbolic sine.

```
: fcosh         \ F: f -- cosh(f) ; (e^x + 1/e^x)/2
```
Floating point hyberbolic cosine.

```
: ftanh         \ F: f -- tanh(f) ; (e^x - 1/e^x)/(e^x + 1/e^x)
```
Floating point hyberbolic tangent.

```
: fasinh        \ F: f -- asinh(f) ; ln(f+sqrt(1+f*f))
```
Floating point hyberbolic arcsine.

```
: facosh        \ F: f -- acosh(f) ; ln(f+sqrt(f*f-1))
```
Floating point hyberbolic arccosine.

```
: fatanh        \ F: f -- atanh(f) ; ln((1+f)/(1-f))/2
```
Floating point hyberbolic arctangent.

14.22 Number conversion

```
: 10**n         \ n -- ; -- f
```
Generate a floating point value 10 to the power n, where n is an integer.

```
: (>FLOAT)      \ c-addr u -- flag ; F: -- f | --
```
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 flag is returned false and the float stack is unchanged.

```
: >FLOAT        \ c-addr u -- flag ; F: -- f | --
```
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 flag is returned false and the float stack is unchanged. Leading and trailing white space are removed before processing. If the resulting string is of zero length, true is returned with a floating point zero. Yes, this is what the standard requires. The previous behaviour without this special case is available as `(>FLOAT)` above.

14.23 FP output

A significant portion of the output code is taken from *FPOUT v3.7* by Ed. See

`http://dxforth.webhop.org/`

The previous code used up to v4.42 is retained in the files *Lib/Ndp387.v442.fth* and *Lib/Hfp387.v442.fth*.

`: precision     \ -- u`

Returns the number of significant digits used by `F.` `FE.` and `FS.`.

`: set-precision \ u --`

Sets the number of significant digits used by `F.` `FE.` and `FS.`.

`: places        \ u --`

Sets the number of significant digits used by `F.` `FE.` and `FS.`. The ANS version of this word is `SET-PRECISION`, which should be used in new code.

`: BadFloat?     \ F: f -- ; -- caddr u true | false`

If the float is a NaN or Infinite, return a string such as "+NaN" and true, otherwise just return false (0).

`: represent     \ c-addr len -- n flag1 flag2 ; F: f --`

Assume that the floating number is of the form +/-0.xxxxEyy. Round the significand xxxxx to *len* significant digits and place its representation at *c-addr*. If *len* is zero round the fractional significand to a whole number. If *len* is negative the fractional significand is rounded to zero. *Flag2* is true if the results are valid. *N* is the signed integer version of yy and *flag1* is true if *f* is negative. In this implementation all errors are handled by exceptions, and so *flag2* is always true except for NaNs and Infinites. The number of characters placed at *c-addr* is the greater of *len* or `MAX-FLOAT-DIGITS`. For a Nan or Infinite, a three character non-numeric string is returned.

`: (FS.)         \ F: f -- ; n -- c-addr u`

Convert float *f* to a string *c-addr/u* in scientific notation with *n* places right of the decimal point.

`: FS.R          \ F: r -- ; n u --`

Display float *f* in scientific notation right-justified in a field width *u* with *n* places right of the decimal point.

`: FS.           \ F: f --`

Display float *f* in scientific notation, with one digit before the decimal point and a trailing space.

`: (FE.)         \ F: r -- ; n -- c-addr u`

Convert float *f* to a string c-addr u in engineering notation with n places right of the decimal point.

`: FE.R          \ F: r -- ; n u --`

Display float *f* in engineering notation right-justified in a field width *u* with *n* places right of the decimal point.

`: FE.           \ F: f --`

Display float *f* in engineering notation, in which the exponent is always a power of three, and the significand is always in the range 1.xxx to 999.xxx.

`: (F.)          \ F: f -- ; n -- c-addr u`

Convert float *f* to string *c-addr/u* in fixed-point notation with *n* places right of the decimal point.

`: F.R              \ F: f -- ; n u --`

Display float *f* in fixed-point notation right-justified in a field width *u* with *n* places right of the decimal point.

`: F.               \ F: f --`

Display f as a float in fixed point notation with a trailing space. The ANS specification says that the display is in fixed-point format, but restricted by `PRECISION`. What should 1e308 display? In this implementation 1e308 displays a 1 followed by 308 zeros. Several people believe that the specification for `F.` is broken. For a display word that always provides sensible output, use `G.` below. Convert float *f* to string *c-addr/u* with *n* places right of the decimal point. Fixed-point is used if the exponent is in the range -4 to 5 otherwise scientific notation is used.

`: G.R              \ F: f -- ; n u --`

Display float *f* right-justified in a field width *u* with *n* places right of the decimal point. Fixed-point is used if the exponent is in the range -4 to 5 otherwise scientific notation is used.

`: G.               \ F: f --`

Display float *f* followed by a space. Floating-point is used if the exponent is in the range -4 to 5 otherwise use scientific notation. Non-essential zeros and signs are removed.

`: f?               \ addr -- ; displays contents of addr`

Displays the contents of the given `FVARIABLE`.

`: f.s              \ F: i*f -- i*f`

Display the contents of the floating point stack in a vertical format.

`: f.sh             \ F: i*f -- i*f`

Display the contents of the floating point stack in a horizontal format.

14.24 Patch FP into the system

`: fnumber?         \ c-addr -- 0 | n 1 | d 2 | -2 ; F: -- r`

Behaves like the integer version of `NUMBER?` except that if integer conversion fails, 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.

`: reals            \ -- ; turn FP system on`

Enables the floating point package for number conversion.

`: integers         \ -- ; turn FP system off`

Disables the floating point package for number conversion.

14.25 PFW2.x compatibility

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

Used in the form `"F# <number>"`, the <number> string is converted and promoted if required to a floating point number. If the system is compiling the float is compiled. If <number> cannot be converted an error occurs.)

14.26 Debugging support

Debugging floating point code is often difficult, as failures can occur because of the necessary approximations involved in floating point operations.

If you set the constant `[FPDEBUG]` true when compiling *Ndp387.fth*, a debug build will be constructed. The state of the FPU will be checked after each word. If a floating point exception has been generated, a diagnostic is issued, and the system aborts. Set this only when testing, as it slows down the normal operation of floating point words.

The debugger works by intercepting the end of each code definition which is finished by `FNEXT,` rather than the normal `NEXT,` or `RET.` See the source code in *\i{Lib\Ndp387.fth for more details.

```
: +fpcheck       \ -- ; enable FP checking
```
Enables the floating point debugger if it has been compiled.

```
: -fpcheck       \ -- ; disable FP checking
```
Disbles the floating point debugger if it has been compiled.

15 Multitasker

15.1 Introduction

The multitasker supplied with VFX Forth is derived from the multitasker provided with the MPE Forth cross compilers, v6.1 onwards. Using a multitasking system can greatly simplify complex tasks by breaking them down into manageable chunks. This chapter leads you through:

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

The multitasker source code is in the file *Lib/Lin32/MultiLin32*. Note that the full version of this file with all switches set except for test code is compiled as part of the second stage build, but is not present by default in the kernel version of VFX Forth.

15.2 Configuration

The configuration of the multitasker is controlled by constants that control what facilities are compiled:

```
0 constant event-handler?   \ true for event handler
0 constant message-handler? \ true for message handler
0 constant semaphores?      \ true for semaphores
0 constant test-code?       \ true for test code
```

15.3 Initialising the multitasker

The multitasker needs to be initialised before use. At compile time you must define the tasks that your system requires and at run-time, all the tasks must be initialised.

Before use the multitasker must be initialised by the word `INIT-MULTI`, which initialises the primary task `MAIN`, and enables the multi-tasker.

To disable the multitasker, use `SINGLE`.

To enable the multitasker, use `MULTI`, which starts the scheduler so new tasks can be added.

15.4 Writing a task

Tasks are very straightforward to write, but the way tasks are scheduled needs to be understood. This implementation uses the Linux **pthreads** API, and so tasks are pre-emptively scheduled. This is different from the cooperative scheduler used by embedded systems. Despite this, the word `PAUSE` which yields a timeslice is retained for compatibility, and `PAUSE` is where the MPE event handling is incorporated.

```
: ACT1ON1     \ -- ; An example task
  TASK0-IO                  \ select the console as the I/O device
  DUP IP-HANDLE !  OP-HANDLE !
  BEGIN                     \ Start an endless loop
    [CHAR] * EMIT           \ Produce a character  )
    1000 ms                 \ Wait 1 second
    PAUSE                   \ Needed!
  AGAIN                     \ Go round again
;
TASK TASK1    \ -- tcb ; name task, get space for it
```

The task name created by `TASK` is used as the task identifier by all words that control tasks.

15.4.1 Task dependent variables

An area of memory known as the `USER` area is set aside for each task. This is often called thread local storage. This memory contains user variables which contain task specific data. For example, the current number conversion radix `BASE` is normally a user variable as it can vary from task to task.

A user variable is defined in the form:

```
n USER <name>
```

where n is the nth byte in the user area. The word `+USER` can be used to add a user variable of a given size:

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

The use of `+USER` avoids any need to know the offset at which the variable starts.

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

15.5 Controlling tasks

Tasks can be controlled in the following ways:
- activated
- halted
- restarted after it has been halted
- terminated.

15.5.1 Activating a task

A task is started by activating it. To activate a task, use `INITIATE`,

```
' <action> <task> INITIATE
```

where `' <action>` gives the xt of the word to be run and `<task>` is the task identifier. The task identifier is used to control the task. Tasks defined by TASK <name> return a task identifier when <name> is executed.

15.5.2 Stopping a task

A task may be temporarily suspended. A task may also halt itself. To temporarily stop a task, use HALT. HALT is used in the form:

```
<task> HALT
```

where <task> is the task to be stopped. To restart a halted task, use RESTART which is used in the form:

```
<task> RESTART
```

where <task> is the task to restart.

To stop the current task (i.e. stop itself) use *\fo{STOP(–).

15.5.3 Terminating a task

Terminating a task halts it, performs an optional clean up action, and calls the operating system thread end function. A thread must terminate itself, which leads to some complexities. However, it does give the task an opportunity to release any resources it may have allocated (especially memory) at start up or during its execution. To terminate a task use:

```
  <task> TERMINATE
```

Before the operating system thread end function is called, the terminating task will execute its clean up code. The XT of the clean up code is held in the task control block. If no clean up action is required, zero is used.

```
  ... ['] CleanUp MyTask AtTaskExit ...
```

If you want a task to be a BEGIN ... UNTIL loop rather than an endless loop, this is perfectly legal, as returning from a thread will call the clean up code and then the ExitThread function. However, you must define an exit code before you return from the task. Note that on entry to the task there will already be a 0 on the stack.

```
: MyTask        \ 0 -- exitcode
  <initialisation>         \ initialise task resources
  begin                    \ round and round until done
    <actions>  MyDone @
  until
  drop 0                   \ paranoid, return 0 as success
;
```

Unlike MPE's embedded systems, under Linux you cannot predict how long a task will take to start after `INITIATE` or shut down after `TERMINATE`.

15.6 Handling messages

An essential feature of the multitasker is the ability to send and receive messages between tasks. For cross compiler compatiblility, the operating system mechanisms are not used.

15.6.1 Sending a message

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

```
message task SEND-MESSAGE
```

where *message* is a 32-bit message and *task* is the identifier of the receiving task. The message can be data, an address or any other type of information but its meaning must be known to the receiving task.

15.6.2 Receiving a message

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

15.7 Events

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

15.7.1 Writing an event

An event is a normal Forth word. An event is associated to a task so that when the event is triggered, the task is resumed. Therefore, an event is usually used as initialisation for a task. Note that an event handler must have NO net stack effect.

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

```
ASSIGN EVENT1 task TO-EVENT
```

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

There are two ways of triggering an event:

- using `SET-EVENT`
- setting a bit in the task status word.

`SET-EVENT` is a word that sets an event flag for a task. Once the event flag is set, the tasker will execute the event before it switches to the task's main-line code. The task is also restarted.

A bit can be set in a task's status word that indicates to the multitasker that an event has taken place. This method can be used to trigger an event from a hardware interrupt or a device driver. Refer to 'The multitasker internals' section later in the chapter for details on the status cell. This mechanism can be used to signal that some event has taken place, and that consequent processing should start.

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

15.8 Critical sections

Sometimes the multitasker has to be inhibited so that other tasks are not run during critical operations that would otherwise cause the scheduler to operate. This is achieved using the words `SINGLE` and `MULTI`. Note that these do *\{not} stop the Linux scheduler, only the MPE extensions. If a full critical section is required, see the semaphore source to find out how to use the Windows critical section API.

```
SINGLE   -- ; inhibit tasker
MULTI    -- ; restart tasker
```

The following words provided for embedded systems have no equivalent because application programs have no direct access to the interrupt control mechanisms:

```
  DI  EI  SAVE-INT  RESTORE-INT  [I  I]
```

15.8.1 Semaphores

A `SEMAPHORE` is a structure used for signalling between tasks, and for controlling resource usage. It has two fields, a counter (cell) and an owner (taskid, cell). The counter field is used as a count of the number of times the resource may be used, and the owner field contains the TCB of the task that last gained access. This field can be used for priority arbitration and deadlock detection/arbitration.

This design of a semaphore can be used either to lock a resource such as a comms channel or disc drive during access by one task, or as a counted semaphore controlling access to a buffer. In the second case the counter field contains the number of times the resource can be used. Semaphores are accessed using `SIGNAL` and `REQUEST`.

`SIGNAL` increments the counter field of a semaphore, indicating either that another item has been allocated to the resource, or it is available for use again, 0 indicating that it is in use by a task.

`REQUEST` waits until the counter field of a semaphore is non-zero, and then decrements the counter field by one. This allows the semaphore to be used as a COUNTED semaphore. For example a character buffer may be used where the semaphore counter shows the number of available characters. Alternatively the semaphore may be used purely to share resources. The semaphore is initialised to one. The first task to `REQUEST` it gains access, and all other tasks must wait until the accessing task `SIGNAL`s that it has finished with the resource.

15.9 Multitasker internals

A multitasker tries to simulate many processors with just one processor. It works by rapidly switching between each task. On each task switch it saves the current state of the processor, and restores the state that the next task needs. The Forth multitasker creates a task control block for each task. The task control block (TCB) is a data structure which contains information relevant to a task.

15.10 A simple example

The following example is a simple demonstration of the multitasker. Its role is to display a hash '#' every so often, but leaving the foreground Forth console running. To use the multitasker you must compile the file *LIB\MULTIWIN32.FTH* into your system. Note that the file has already been compiled by the Studio IDE in *VfxForth.exe*, but is not present in *VfxBase.exe*.

The following code defines a simple task called `TASK1`. It displays a '$' character every second.

```
VARIABLE DELAY     \ time delay between #'s in milliseconds
  1000 DELAY !             \ initialise time delay
: ACTION1          \ -- ; task to display #'s
  TASK0-IO                 \ select the console as the I/O device
  DUP IP-HANDLE !  OP-HANDLE !
  [CHAR] $ EMIT            \ Display a dollar
  BEGIN                    \ Start continuous loop
    [CHAR] # EMIT          \ Display a hash
    DELAY @ MS             \ Reschedule Delay times
    PAUSE                  \ At least one per loop
  AGAIN                    \ Back to the start ...
;
```

The use of `PAUSE` in this example is not actually required as `MS` periodically calls `PAUSE`.

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

```
INIT-MULTI
```

The word `INIT-MULTI` initialises all the multitasker's data structures and starts multitasking. This word need only be executed once in a multitasking system and is usually executed at start up.

Note that on entry to a task, the stack depth will be 1. This happens because Linux requires a return value when a task terminates, and a value of zero is provided by the task initialisation code.

To run the example task, type:

```
TASK TASK1
ASSIGN ACTION1 TASK1 INITIATE
```

This will activate ACTION1 as the action of task TASK1. Immediately you will see a dollar and a hash displayed. If you press <return> a few times, you notice that the Forth interpreter is still running. After a few seconds another hash character will appear. This is the example task working in the background.

The example task can be controlled in several ways:

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

Changing the variable DELAY can change the rate of production of hashes. Try:

```
2000 DELAY !
```

This changes the number of milliseconds between displaying hashes to 2000 milliseconds. Therefore the rate of displaying hashes halves.

Typing the task name followed by HALT halts the task:

```
TASK1 HALT
```

You notice that the hashes are not displayed any more.

The task is restarted by RESTART. Type:

```
TASK1 RESTART
```

You notice that the hashes are displayed again.

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

```
TASK1 TERMINATE
ASSIGN ACTION1 TASK1 INITIATE
```

You notice the dollar and the hash are displayed, followed by more hashes.

15.11 Glossary

15.11.1 Configuration

`1 constant event-handler?        \ -- n`

The event handling code will be compiled if this constant is true.

```
1 constant message-handler?     \ -- n
```
The message handling code will be compiled if this constant is true.

```
1 constant semaphores?          \ -- n
```
The semaphore code will be compiled if this constant is true.

```
0 constant test-code?           \ -- n
```
The test code will be compiled if this constant is true.

15.11.2 Structures and support

```
#38 constant /tcb.callback      \ -- len
```
Size of the task callback data and code. Used for error checks. X86 version.

```
#44 constant /tcb.callback      \ -- len
```
Size of the task callback data and code. Used for error checks. ARM version.

```
struct /TCB     \ -- size
```
Returns the size of a TCB structure, which controls the task.

```
  int tcb.link          \ link to next task ; MUST BE FIRST      0
  int tcb.hthread       \ task handle                            4
  int tcb.up            \ user pointer                           8
  int tcb.pumpxt        \ xt of message pump or 0 for none      12
  int tcb.status        \ status bits                           16
  int tcb.mesg          \ message from another task             20
  int tcb.msrc          \ TCB of task from which message came   24
  int tcb.event         \ xt of event handler                   28
  int tcb.clean         \ xt of clean up handler                32
  /sem_t field tcb.haltsem \ sem_t (16) for halt/suspend        36
  /tcb.callback field tcb.callback \ task callback structure    52
  aligned               \ force to cell boundary
end-struct
```

The task status cell reserves the low 8 bits for use by VFX Forth. The other bits may be used by your application.

```
Bit    When set                        When Reset
0      RFU                             RFU
1           Message pending but not read   No messages
2           Event triggered                No events
3           Event handler has been run     No events - reset by user
4..7   RFU                             RFU
8..31  User defined                    User defined
```

```
cell +USER ThreadExit?  \ -- addr
```
Holds a non-zero value to cause the thread to exit.

```
cell +USER ThreadTCB    \ -- addr
```
Holds a pointer to the thread's TCB.

```
cell +USER ThreadSync   \ -- addr
```
Holds bit patterns used for intertask synchronisation. See later section.

```
: AtTaskExit    \ xt tcb -- ; set task exit action
```

Sets the given task's cleanup action. Use in the form:

```
' <action> <task> AtTaskExit
```

`: perform       \ addr --`
Execute contents of addr if non-zero. The non-zero contents of *addr* are EXECUTEd.

15.11.3 Task definition and access

`create main     \ -- addr ; tcb of main task`
The task structure for the first task run, usually the console or the main application.

`: InitTCB       \ addr --`
Initialise a task control block at addr. X86 version.

`: InitTCB       \ addr --`
Initialise a task control block at addr. ARM version.

`: task          \ -- ; -- addr ; define task, returns TCB address`
Use in the form TASK <name>, creates a new task data structure called <name> which returns the address of the data structure when executed.

`: Self          \ -- tcb|0 ; returns TCB of current task`
Returns the task id (TCB) of the current task. If called outside a task, zero is returned.

`: his           \ task uservar -- addr`
Given a task id and a USER variable, returns the address of that variable in the given task. This word is used to set up USER variables in other tasks. Note that the task must be running.

15.11.4 Task handling primitives

`0 value multi?  \ -- flag`
Returns true if the tasker is enabled.

`: single        \ -- ; disable scheduler`
Disables the Forth portions of the scheduler, but does not disable Linux scheduling.

`: multi         \ -- ; enable scheduler`
Enables the Forth portions of the scheduler, but does not disable Linux scheduling.

`defer pause     \ --`
PAUSE is the software entry to the pre-emptive scheduler, and should be called regularly by all tasks. The phrase sched_yield drop occurs at the end of the default action (PAUSE). If the task needs more than this and does not use one of the existing message loop words such as IDLE, place the XT of the message pump word in offset TCB.PUMPXT of the Task Control Block and that XT will be called once every time PAUSE is called. Because of the way Linux works, PAUSE also controls task closure. A task that does not call PAUSE cannot be safely terminated except by the task itself, or by a call to the API function kill(). A task that calls PAUSE in a loop without calling any delay mechanism will cause CPU hogging.

`: (pause)       \ -- ; the scheduler itself`
The action of PAUSE after the multitasker has been compiled. If SINGLE has been set, no action is taken. If PAUSE was not called from a task and MULTI is set, the action is **sched_yield**.

`: restart       \ tcb -- ; mark task TCB as running`
If the task has been initiated but is now HALTed or STOPped, it will be restarted.

`: halt          \ tcb -- ; mark thread as halted`
Stops an INITIATEd task from running until RESTART is used.

`: stop          \ -- ; halt oneself`

`HALT`s the current task.

`: running?      \ tcb -- u`

Returns the task's semaphore value, where non-zero indicates that it is running. Returns 0 on error.

15.11.5 Event handling

`: set-event     \ task -- ; set event trigger in task TCB`

Sets the event trigger bit in the task. When `PAUSE` is next executed by that task, its event handler will be run.

`: event?        \ task -- flag ; true if task had event`

Returns true if the task has received an event trigger, but has not yet run the event handler.

`: clr-event-run \ -- ; reset own EVENT_RUN flag`

Clear the `EVENT_RUN` flag of the current task. This is usually done if the task has to be put back to sleep after the event handler has been run.

`: to-event      \ xt task -- ; define action of a task`

Used in the form below to define a task's event handler:

```
assign <action> <task> to-event
```

15.11.6 Message handling

`: msg?          \ task -- flag ; true if task has message`

Returns true if the given task has received a message.

`: send-message  \ mesg task -- ; send message to task (wakes it up)`

Sends a message to a task, waking it up if it was asleep. Interpretation of a message is the resposibility of the receiving task. If the receiving task has unprocessed messages, the sending task blocks.

`: get-message   \ -- mesg task ; wait for any message`

Waits until a message has been received from another task. Interpretation of a message is the resposibility of the receiving task. See `MSG?` which tells you if a message is available.

`: wait-event/msg        \ -- ; wait for message or event trigger`

Wait until a message or event occurs.

15.11.7 Task management

`: to-task       \ xt task -- ; set action of task`

Used in the form below to define a task's action:

```
assign <action> <task> to-task
```

`: to-pump       \ xt task -- ; set message loop of task`

Used in the form below to define rhe action of the message pump:

```
assign <action> <task> to-pump
```

`: initiate      \ xt task -- ; start task from scratch`

Initialises a task running the given xt. All required O/S resources are allocated by this word.

`: terminate     \ task -- ; stop task, and remove from list`

Causes the specified task to die. You should not make assumptions as to how long this will take. Unlike the embedded systems implementations, this word is very operating system dependent. The task may still be alive on return from this call.
N.B. Do not use `self terminate` to cause a task to end. Use the following instead:

```
: Suicide    \ -- ; terminate current task
  pause  termThread  0 pthread_exit
;

... Suicide
```

`: start:        \ task -- ; exits from caller`

START: is used inside a colon definition. The code before START: is the task's initialisation, performed by the current task. The code after START: up to the closing ; is the action of the task. For example:

```
  TASK FOO
  : RUN-FOO
    ...
    FOO START:
    ...
    begin ... pause again  )
  ;
```

All tasks must run in an endless loop, except for initialisation code. There are exceptions to this, and these are discussed in the section on terminating a task. When RUN-FOO is executed, the code after START: is set up as the action of task FOO and started. RUN-FOO then exits. If you want to perform additional actions after starting the task, you should use IINITIATE to start the task.

`: TaskState     \ task -- state`

Returns true if the task has started and zero if the thread has finished.

`: init-multi    \ -- ; initialisation of multi-tasking`

Initialise the Forth multitasker to a state where only the task MAIN is known to be running. INIT-MULTI is added to the cold chain and is also called during compilation of *MultiLin32.fth*. This word **must** be run from MAIN.

`: term-multi    \ --`

Performed in the exit chain when the program terminates. closes all active tasks except SELF. This allows all task clean-up actions to be performed before the program itself finishes.

`: .task         \ task -- ; display task name`

Given a task, e.g. as returned by SELF, display its name or address.

`: .tasks        \ -- ; display active tasks`

Display a list of all the active Forth tasks.

15.11.8 Task synchronisation

`$AAAA5555 constant TaskReady    \ -- n`

At task initiation, USER variable THREADSYNC is set to zero. Set THREADSYNC to this value to indicate that the task is willing to synchronise with another task.

`$5555AAAA constant TaskReadied  \ -- n`

A synchronising task sets another task's THREADSYNC to this value to indicate that synchronisation is complete.

`: WaitForSync   \ --`

Perform the slave synchronisation sequence.

`: [Sync         \ task -- task`

Used by a master task in the form:

```
[Sync ... Sync]
```

to synchronise and pass data to another task, usually when `USER` variables must be initialised. The slave task must execute `WAITFORSYNC`.

`: Sync]          \ task --`
Used by a master task in the form:

```
[Sync ... Sync]
```

to indicate the end of synchronisation.

15.11.9 Semaphores

`struct /semaphore       \ -- len`
Structure used for Linux i32 semaphores.

`: semaphore     \ -- ; -- addr [child]`
A `SEMAPHORE` is an extended variable used for signalling between tasks and for resource allocation. The counter field is used as a count of the number of times the resource may be used, and the arbiter field contains the TCB of the task that last gained access. This field can be used for priority arbitration and deadlock detection/arbitration.

`: InitSem       \ semaphore --`
Initialise the semaphore. This **must** be done before using it.

`: ShutSem       \ semaphore --`
Delete the critical section associated with the smaphore.

`: LockSem       \ semaphore --`
Lock the semaphore.

`: UnlockSem     \ semaphore --`
Unlock the semaphore.

`: signal        \ sem -- ; increment counter field of semaphore,`
`SIGNAL` increments the counter field of a semaphore, indicating either that another item has been allocated to the resource, or that it is available for use again, 0 indicating in use by a task.

`: request       \ sem -- ; get access to resource, wait if count = 0`
`REQUEST` waits until the counter field of a semaphore is non-zero, and then decrements the counter field by one. This allows the semaphore to be used as a **counted** semaphore. For example a character buffer may be used where the semaphore counter shows the number of available characters. Alternatively the semaphore may be used purely to share resources. The semaphore is initialised to one. The first task to `REQUEST` it gains access, and all other tasks must wait until the accessing task `SIGNAL`s that it has finished with the resource.

16 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 and with the code supplied with MPE's embedded targets. This code assumes the presence of `TICKS` which returns a time value incremented in milliseconds.

The timebase is approximate, and granularity and jitter are affected by Linux and the time taken to run your own code. By default, the timer is set to run every 100ms. The source code is in the the file *Lib/Lin32/TimeBase.fth*. If you are using the multitasker, you **must** compile *TimeBase.fth* after *MultiLin32.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 VFX Forth, these time periods must be less than 2^31-1 milliseconds, say 596 hours or 24 days, whereas if the code is ported to a 16 bit system, time periods must be less than 2^15-1 milliseconds, say 32 seconds.

16.1 The basics of timers

These basic words are defined for applications to use the

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

```
start-timers
: t     \ -- ; will run every 2 seconds
  [char] * emit
;
' t 2 seconds every     \ returns timerid, use TSTOP to stop it
```

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

`AFTER` is useful for creating timeouts, such as is 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.

16.2 Considerations when using timers

All timers are executed within a single callback, 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 callback that handles all the timers sets `IP-HANDLE` and `OP-HANDLE` to a default that corresponds to the interactive Forth console. If you use Forth I/O words such as `EMIT` and `TYPE` within a timer action, you **must** set `IP-HANDLE` and `OP-HANDLE` 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 `IP-HANDLE` and `OP-HANDLE` 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.

Be sure to use `START-TIMERS` in your main task, so that the timer is not destroyed if a thread terminates.

16.3 Implementation issues

The following discussion is relevant if you want to modify this code or port it to an embedded target. Functionally equivalent code is provided with MPE's Forth VFX cross compilers. In the Linux environment, timer interrupts are implemented by signals, callbacks and critical sections.

By default, the word `DO-TIMERS` is run from within the periodic timer callback. You may have latency issues if a large number of timers is used, or if some timer routines take a considerable time. In this case, it may be 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 timer, say 1ms, for a clock, and to trigger `TIMER-TASK` every say 32 ms.

16.4 Timebase glossary

`/sem_t buffer: lpcs     \ -- sem`

Semaphore used for critical sections.

`#32 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 below.

`struct itimer   \ -- len`

Interval timer structure.

```
  cell field itlink                     \ link to next timer ; MUST be first
  cell field itTimerId                  \ timer ID
  cell field itinterval                 \ period of timer in MS
  cell field ittimeout                  \ next timeout
  cell field itmode                     \ mode/flags, 0=periodic, 1=one shot
  cell field itxt                       \ word to execute
end-struct
```

`: 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 ; 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.

`: seconds       \ n -- n'`
Converts seconds to milliseconds.

`struct /itimerval       \ -- len`
Corresponds to the Linux **itimerval** structure.

```
  int it.currsecs       \ current period
  int it.currusecs      \ in seconds & microseconds
  int it.nextsecs       \ next period (0 to stop)
  int it.nextusecs      \ in seconds & microseconds
end-struct
```

`/itimerval buffer: TBtimer      \ -- addr`
Structure controlling the main timer.

`: SetTimerData  \ ms timer --`
Set a Linux **itimerval** structure to run every *ms* milliseconds. Setting 0 ms will stop the timer.

`#100 constant /period   \ -- ms`
Main timer period in milliseconds.

`3 0 callback: SIGALRMhandler    \ -- entrypoint`
SIGALRM callback for main timer.

`: doSIGALRM     \ signum *siginfo *ucontext --`
Action of main timer.

`: start-timers  \ -- ; Start internal time clock`
Initialises the timebase system, and starts the timebase system. Note that all timer actions are cleared. Performed at start up.

`: Stop-Timers   \ -- ; disable timer system`
Halts the timebase interrupt.

17 A BNF Parser in Forth

17.1 Introduction

Backus-Naur Form, BNF, is a notation for the formal description of programming languages. While most commonly used to specify the syntax of "conventional" programming languages such as Pascal and C, BNF is also of value in command language interpreters and other language processing.

This paper describes a Forth extension which transforms BNF expressions to executable Forth words. This gives Forth a capability equivalent to YACC or TMG, to produce a working parser from a BNF language description.

This article first appeared in ACM SigFORTH Newsletter vol. 2 no. 2. Since then the code has been updated from the original by staff at MPE, and this documentation has been derived from the article supplied by Brad Rodriguez, whose original implementation is a model of Forth programming.

The source code is compiled by the second stage build and is in the file SOURCES\VFXBASE\BNF.FTH.

17.2 BNF Expressions

BNF expressions or productions are written as follows:

```
production  ::=  term ... term    \ alternate #1
              |  term ... term    \ alternate #2
              |  term ... term    \ alternate #3
```

This example indicates that the given production may be formed in one of three ways. Each alternative is the concatenation of a series of terms. A term in a production may be either another production, called a nonterminal, or a fundamental token, called a terminal.

A production may use itself recursively in its definition. For example, an unsigned integer can be defined with the productions

```
<digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
<number> ::= <digit> | <digit> <number>
```

which says that a number is either a single digit, or a single digit followed by another number of one or more digits.

We will use the conventions of a vertical bar | to separate alternatives, and angle brackets to designate the name of a production. Unadorned ASCII characters are terminals, the fundamental tokens.

17.3 A Simple Solution through Conditional Execution

The logic of succession and alternation can be implemented in two "conditional execution" operators, && and ||. These correspond exactly to the "logical connectives" of the same names in the C language (although their use here was actually inspired by the Unix "find" command). They are defined:

```
: || IF R> DROP 1 THEN ;    ( exit on true)
: && 0= IF R> DROP 0 THEN ; ( exit on false)
```

|| given a true value on the stack, exits the colon definition immediately with true on the stack. This can be used to string together alternatives: the first alternative which is satisfied (returns true) will stop evaluation of further alternatives.

&& given a false value on the stack, exits the colon definition immediately with false on the stack. This is the "concatenation" operator: the first term which fails (returns false) stops evaluation and causes the entire sequence to fail.

We assume that each "token" (terminal) is represented by a Forth word which scans the input stream and returns a success flag. Productions (nonterminals) which are built with such tokens, ||, and &&, are guaranteed to return a success flag.

So, assuming the "token" words '0' thru '9' have been defined, the previous example becomes:

```
: <DIGIT>   '0' || '1' || '2' || '3' || '4'
               || '5' || '6' || '7' || '8' || '9' ;
: <NUMBER1>   <DIGIT> && <NUMBER> ;
: <NUMBER>    <DIGIT> || <NUMBER1> ;
```

Neglecting the problem of forward referencing for the moment, this example illustrates three limitations:

a) we need an explicit operator for concatenation, unlike BNF.

b) && and || have equal precedence, which means we can't mix && and || in the same Forth word and get the equivalent BNF expression. We needed to split the production <NUMBER> into two words.

c) we have made no provision for restoring the scan pointer if a BNF production fails.

We will address these next.

17.4 A Better Solution

Several improvements can be made to this "rough" BNF parser, to remove its limitations and improve its "cosmetics."

a) Concatenation by juxtaposition. We can cause the action of && to be performed "invisibly" by enforcing this rule for all terms (terminals and nonterminals): Each term examines the stack on entry. if false, the word exits immediately with false on the stack. Otherwise, it parses and returns a success value.

To illustrate this: consider a series of terms

```
<ONE> <TWO> <THREE> <FOUR>
```

Let <ONE> execute normally and return "false." The <TWO> is entered, and exits immediately, doing nothing. Likewise, <THREE> and <FOUR> do nothing. Thus the remainder of the expression is skipped, without the need for a return-stack exit.

b) Precedence. By eliminating the && operator in this manner, we make it possible to mix concatenation and alternation in a single expression. A failed concatenation will "skip" only as far as the next operator. So, our previous example becomes:

```
: <NUMBER> <DIGIT> || <DIGIT> <NUMBER> ;
```

c) Backtracking. If a token fails to match the input stream, it does not advance the scan pointer. Likewise, if a BNF production fails, it must restore the scan pointer to the "starting point" where the production was attempted, since that is the point at which alternatives must be tried. We therefore enforce this rule for all terminals and nonterminals: Each term saves the scan pointer on entry. If the term fails, the scan pointer is restored; otherwise, the saved value is discarded.

We will later find it useful to "backtrack" an output pointer, as well.

d) Success as a variable. An examination of the stack contents during parsing reveals the surprising fact that, at any time, there is only one success flag on the stack! (This is because flags placed on the stack are immediately "consumed.") We can use a variable, SUCCESS, for the parser success flags, and thereby simplify the manipulations necessary to use the stack for other data. All BNF productions accept, and return, a truth value in success.

17.5 Notation

<BNF is used at the beginning of a production. If SUCCESS is false, it causes an immediate exit. Otherwise, it saves the scan pointer on the return stack.

BNF> is used at the end of a production. If SUCCESS is false, it restores the scan position from the saved pointer. In any case, it removes the saved pointer from the return stack.

<BNF and BNF> are "run-time" logic, compiled by the words ::= and ;; respectively.

::= name starts the definition of the BNF production name.

;; ends a BNF definition.

| separates alternatives. If SUCCESS is true, it causes an immediate exit and discards the saved scan pointer. Otherwise, it restores the scan position from the saved pointer.

{ ... } denotes a production statement which is issued when successful evaluation of the preceding checks has been performed.. The alternative form { _ }{ _ } allows conditional productions in which the first part is produced when SUCCESS is true, and the second part is produced when SUCCESS is false.

[[...]] denotes a block that must be performed 0 or more times, and thus is totally optional. Alternatives are not permitted.

<< ... >> denotes a block that must be performed at least once. Alternatives are not permitted.

There are four words which simplify the definition of token words and other terminals:

@TOKEN fetches the current token from the input.

+TOKEN advances the input scan pointer.

=TOKEN compares the value on top of stack to the current token, following the rules for BNF parsing words.

nn TOKEN name builds a "terminal" name, with the ASCII value nn.

The parser uses the Forth >IN as the input pointer, and the dictionary pointer DP as the output pointer. These choices were made strictly for convenience; there is no implied connection with the Forth compiler.

17.6 Examples and Usage

The syntax of a BNF definition in Forth resembles the "traditional" BNF syntax:

Traditional:

```
prod ::= term term | term term
```

Forth:

```
::= prod term term | term term ;;
```

The first example below is a simple pattern recognition problem, to identify text having balanced left and right parentheses. Several aspects of the parser are illustrated by this example:

a) Three tokens are defined on line 4. To avoid name conflicts, they are named with enclosing quotes. <EOL> matches the end-of-line character in the Forth Terminal Input Buffer.

b) A recursive production, <S>, is shown.

c) The definition of <S> also shows a null alternative. This is often encountered in BNF. The null alternative parses no tokens, and is always satisfied.

d) Not all parsing words need be written as BNF productions. The definition of <CHAR> is Forth code to parse any ASCII character, excluding parentheses and nulls. Note that ::= and ;; are used, not to create a production, but as an easy way to create a conditionally-executing (per SUCCESS) Forth word.

e) PARSE shows how to invoke the parser: SUCCESS is initialized to "true," and the "topmost" BNF production is executed. on its return, SUCCESS is examined to determine the final result.

f) PARSE also shows how end-of-input is indicated to the BNF parser: the sequence is defined as the desired BNF production, followed by end-of-line.

The second example parses algebraic expressions with precedence. This grammar is directly from [AH077], p. 138. The use of the productions <T’> and <E’> to avoid the problem of left-recursion is described on p. 178 of that book. Note also:

a) <DIGIT> is defined "the hard way." It would be better to do this with a Forth word.

b) <ELEMENT> requires a forward reference to <EXPRESSION>. We must patch this reference manually.

The third example shows how this algebraic parser can be modified to perform code generation, coincident with the parsing process. Briefly: each alternative of a BNF production includes Forth code to compile the output which would result from that alternative. If the alternative succeeds, that output is left in the dictionary. If it fails, the dictionary pointer is "backtracked" to remove that output. Thus, as the parser works its way, top-down, through the parse tree, it is constantly producing and discarding trial output.

This example produces Forth source code for the algebraic expression.

a) The word ," appends a text string to the output.

b) We have chosen to output each digit of a number as it is parsed. (DIGIT) is a subsidiary word to parse a valid digit. <DIGIT> picks up the character from the input stream before it is parsed, and then appends it to the output. If it was not a digit, SUCCESS will be false and ;BNF will discard the appended character.

If we needed to compile numbers in binary, <NUMBER> would have to do the output. <NUMBER> could start by placing a zero on the stack as the accumulator. <DIGIT> could augment this value for each digit. Then, at the end of <NUMBER>, the binary value on the stack could be output.

c) After every complete number, we need a space. We could factor <NUMBER> into two words, like <DIGIT>. But since <NUMBER> only appears once, in <ELEMENT>, we append the space there.

d) In <PRIMARY>, MINUS is appended after the argument is parsed. In <FACTOR>, POWER is appended after its two arguments are parsed. <T’> appends * or / after the two arguments, and likewise <E’> appends + or -.

In all of these cases, an argument may be a number or a sub-expression. If the latter, the entire code to evaluate the sub-expression is output before the postfix operator is output. (Try it. It works.)

e) PARSE has been modified to TYPE the output from the parser, and then to restore the dictionary pointer.

17.7 Cautions

This parser is susceptible to the Two Classic Mistakes of BNF expressions. Both of these cautions can be illustrated with the production <NUMBER>:

```
::= <NUMBER>
    <DIGIT> <NUMBER> | <DIGIT> ;;
```

a) Order your alternatives carefully. If <NUMBER> were written

```
::= <NUMBER>
<DIGIT> | <DIGIT> <NUMBER> ;;
```

then all numbers would be parsed as one and only one digit! This is because alternative #1 – which is a subset of alternative #2 – is always tested first. In general, the alternative which is the subset or the "easier to-satisfy" should be tested last.

b) Avoid "left-recursion." If <NUMBER> were written

```
::= <NUMBER>
    <NUMBER> <DIGIT> | <DIGIT> ;;
```

then you will have an infinite recursive loop of <NUMBER> calling <NUMBER>! To avoid this problem, do not make the first term in any alternative a recursive reference to the production being defined. (This rule is somewhat simplified; for a more detailed discussion of this problem, refer to [AH077], pp. 177 to 179.)

17.8 Comparison to "traditional" work

In the jargon of compiler writers, this parser is a "top-down parser with backtracking." Another such parser, from ye olden days of Unix, was TMG. Top-down parsers are among the most flexible of parsers; this is especially so in this implementation, which allows Forth code to be intermixed with BNF expressions.

Top-down parsers are also notoriously inefficient. Predictive parsers, which look ahead in the input stream, are better. Bottom-up parsers, which move directly from state to state in the parse tree according to the input tokens, are better still. Such a parser, YACC (a table-driven LR parser), has entirely supplanted TMG in the Unix community.

Still, the minimal call-and-return overhead of Forth alleviates the speed problem somewhat, and the simplicity and flexibility of the BNF Parser may make it the parser of choice for many applications. Experience at MPE shows that BNF parsers are actually quite fast.

17.9 Applications and Variations

Compilers. The obvious application of a BNF parser is in writing translators for other languages. This should certainly strengthen Forth's claim as a language to write other languages.

Command interpreters. Complex applications may have an operator interface sufficiently complex to merit a BNF description. For example, this parser has been used in an experimental lighting control system; the command language occupied 30 screens of BNF.

Pattern recognition. Aho & Ullman [AH077] note that any construct which can be described by a regular expression, can also be described by a context-free grammar, and thus in BNF. [AH077] identifies some uses of regular expressions for pattern recognition problems; such problems could also be addressed by this parser.

An extension of these parsing techniques has been used to implement a Snobol4-style pattern matcher [ROD89a].

Goal directed evaluation. The process of searching the parse tree for a successful result is essentially one of "goal-directed evaluation." Many problems can be solved by goal-directed techniques.

For example, a variation of this parser has been used to construct an expert system [ROD89b].

17.10 References

[AH077] Alfred Aho and Jeffrey Ullman, Principles of Compiler Design, Addison-Wesley, Reading, MA (1977), 604 pp.

[ROD89a] B. Rodriguez, "Pattern Matching in Forth," presented at the 1989 FORML Conference, 14 pp.

17.11 Example 1 - balanced parentheses

```
\ Example #1 - from Aho & Ullman, Principles of Compiler Design, p137
\ This grammar recognises strings having balanced parentheses

hex

ascii ( token '('
ascii ) token ')'
0 token <eol>

::= <char>
  @token
  dup 02A 07F within?
  swap 1 027 within?  or
  dup success !
  +token
;;

::= <s>
        '(' <s> ')' <s>
     |  <char> <s>
     |
;;

: parse
  1 success !
  <s> <eol>
  cr  success @
  if    ." Successful
  else  ." Failed"
  endif
;
```

17.12 Example 2 - Infix notation

```
ascii + token '+'       ascii - token '-'
ascii * token '*'       ascii / token '/'
ascii ( token '('       ascii ) token ')'

ascii ^ token '^'

ascii 0 token '0'       ascii 1 token '1'
ascii 2 token '2'       ascii 3 token '3'
ascii 4 token '4'       ascii 5 token '5'
ascii 6 token '6'       ascii 7 token '7'
ascii 8 token '8'       ascii 9 token '9'

0 token <eol>

::= <digit>
          '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9'
;;

::= <number>
          <digit> <number>
        | <digit> { }
;;

defer <expression>              \ needed for a recursive definition

::= <element>
          '(' <expression> ')'
        | <number>
;;

::= <primary>
          '-' <primary>
        | <element>
;;

::= <factor>
          <primary> '^' <factor>
        | <primary>
;;

::= <t'>
          '*' <factor> <t'>
        | '/' <factor> <t'>
        |
;;
```

```
::= <term>
          <factor> <t'>
;;

::= <e'>
          '+' <term> <e'>
        | '-' <term> <e'>
        |
;;

::= <<expression>>
          <term> <e'>
;;
assign <<expression>> to-do <expression>

: parse
  1 success !
  <expression> <eol>
  cr  success @
  if    ." Successful
  else  ." Failed"
  endif
;
```

17.13 Example 3 - infix notation again with on-line calculation

```
: x^y           \ x y -- n
  dup 0< abort" can't deal with negative powers"
  1 swap 0                            \ -- x 1 y 0
  ?do  over *  loop                   \ -- x x^i
  nip                                 \ -- x^y
;

1 constant mul
2 constant div
3 constant add
4 constant sub

: dyadic        \ x op y -- n
  case  swap
    mul of  *  endof
    div of  /  endof
    add of  +  endof
    sub of  -  endof
            1 abort" invalid operator"
  endcase
;

decimal

ascii + token '+'       ascii - token '-'
ascii * token '*'       ascii / token '/'
ascii ( token '('       ascii ) token ')'

ascii ^ token '^'

ascii 0 token '0'       ascii 1 token '1'
ascii 2 token '2'       ascii 3 token '3'
ascii 4 token '4'       ascii 5 token '5'
ascii 6 token '6'       ascii 7 token '7'
ascii 8 token '8'       ascii 9 token '9'

bl token <sp>
9 token <tab>
0 token <eol>

::= <whitespacechar>    \ -- ; could be expanded to refill input buffer
  <sp> | <tab>
;;

::= <whitespace>
  [[ <whitespacechar> ]]
;;
```

```
::= <digit>
          '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9'
;;

::= <number>
          { 0 }                                \ initial accumulator
          <whitespace>
          << <digit>
             { 10 *  last-token @ ascii 0 -  +  }
          >>
;;

defer <expression>              \ needed for a recursive definition

::= <element>
          '(' <expression> ')'
        | <number>
;;

::= <primary>
          '-' <primary>  { negate }
        | <element>
;;

::= <factor>
          <primary> '^' <factor> { x^y }
        | <primary>
;;

::= <term-op>
          '*' { mul }
        | '/' { div }
;;

::= <term>
          <factor> [[ <whitespace> <term-op> <factor> { dyadic } ]]
;;

::= <exp-op>
          '+' { add }
        | '-' { sub }
;;

::= <<expression>>
          <term> [[ <whitespace> <exp-op> <term> { dyadic } ]]
;;  assign <<expression>> to-do <expression>

: parse
  1 success !
  <expression> <whitespace> <eol>
  cr  success @
  if    ." Successful" cr ." Result = " .
  else  ." Failed"
  endif
;
```

17.14 Acknowledgements

This article first appeared in ACM SigFORTH Newsletter vol. 2 no. 2. Since then the code has been updated from the original by staff at MPE.

Bradford J. Rodriguez

T-Recursive Technology

115 First St. #105

Collingwood, Ontario L9Y 4W3 Canada

bj@forth.org

17.15 Glossary

`variable success        \ -- addr`

This variable is set true if the last BNF statement succeded, otherwise it is false.

`variable skipspace      \ -- addr`

Controls space skipping. When set true, following spaces are skipped.

`variable BNF-ignore-lines       \ -- addr`

Controls line break handling. When set true, line breaks are ignored by REFILLing the input buffer.

`: |             \ -- ; performs OR function`

Performs the OR function inside a BNF definition.

`: ?bnf-error    \ --`

Produce an error message on parsing failure.

`: save-success  \ -- ; R: -- success`

Save the SUCCESS flag on the return stack.

`: check-success \ -- ; R: success --`

Generate an error if the value of SUCCESS previously saved on the return stack was true but now isn't. Useful to provide sensible source error messages inside deeply nested definitions.

`: ::=           \ -- sys ; defines a BNF definition`

Start a BNF definition of the form:

`::= <name> ... ;;`

`: ;;            \ sys -- ; marks end of ::= <name> ... ;; definition`

Ends a BNF definition of the form:

`::= <name> ... ;;`

`: {             \ -- sys`

Marks the tart of production output if SUCCESS is true. Use in the form: "`{` ... `}{` ... `}`" which generates the code for "SUCCESS `@` IF ... ELSE ... THEN".

`: }{            \ sys -- sys'`

Allows an ELSE clause for production output.

: } \ sys --

End of production output

: [[\ -- addr1 addr2 ; start of [[...]] block, loop end inline

Starts an optional block (0 or more repetitions) of the form:

```
[[ ... ]]
```

Note that alternatives using | are not permitted.

:]] \ addr1 addr2 -- ; end of [[...]] block, loop start inline

Ends an optional block of the form:

```
[[ ... ]]
```

Note that alternatives using | are not permitted.

: << \ -- addr1 addr2 ; start of << ... >> block, loop end inline

Starts a block (1 or more repetitions) of the form:

```
<< ... >>
```

Note that alternatives using | are not permitted.

: >> \ addr1 addr2 -- ; end of [[...]] block, loop start inline

Ends a block (1 or more repetitions) of the form:

```
<< ... >>
```

Note that alternatives using | are not permitted.

variable last-token \ -- addr

Holds the last token (or 0) retrieved by @TOKEN.

: token \ n --

Use in the form "<char> TOKEN <name>" to define a word <name> which succeeds if the next token (character) is <char>.

: +spaces \ --

Enables space skipping. If +SPACES does not call nextNonBL then it has to appear BEFORE the last word for which spaces will not be skipped, which is confusing. This way the final word which does not discard its following spaces appears in the source code before the +SPACES, which looks more logical.

: -spaces \ --

Disables space skipping.

: string \ -- ; string <name> text ; e.g. string 'CAP' WS_CAPTION

Used in the form "STRING <name> text" to create a word <name> which succeeds when space delimited text is next in the input stream. Note that text may not contain spaces. Because of some parsing requirements, e.g. some BASICs and FORTRAN, a superset of text will succeed, leaving the residue in the input stream. This means that for "STRING <name> abcd" the strings "a", "ab", and "abc" will also succeed. Thus if you need to test a set of strings, you should test the longest first, e.g:

```
String str1 abcd
String str2 abc
String str3 ab
\ WRONG because abcd will match str3
::= test  str3 | str2 | str1  ;;
\ RIGHT
::= test  str1 | str2 | str3  ;;
```

17.16 Error reporting

Because the BNF parser is a top-down recursive descent parser, when a rule fails, it backtracks to the previous successful position, both in terms of output and source file position. Because of this, the reported error position may be some way before the actual location that triggered the error.

18 Text macro substitution

18.1 Usage

VFX Forth implements text macro substitution, where a text macro named `FOO` my be substituted in a string. When referenced in a string the macro name must be surrounded by % characters. If a % character is needed in a string it must be entered as %%.

Thus if `FOO` is defined as "c:\apps\vfxforth" then the string

```
"Error in file %FOO%\myfile.fth at line "
```

would be expanded to

```
"Error in file c:\apps\vfxforth\myfile.fth at line "
```

Macros are defined in the `Substitutions` vocabulary which is searched when the string is expanded. When executed these words return the address of a counted string for the text to substitute.

`TextMacro: <name>` defines an empty macro with a 255 character buffer in the `Substitutions` vocabulary.

`<string> SETMACRO <name>` sets the given string into the required macro `<name>`. If `<name>` does not exist in the `Substitutions` vocabulary an error is reported. `SETMACRO` may also be used in colon definitions, providing that the macro name already exists. If a colon definition needs to create a new macro name it should use `$SETMACRO` instead.

```
TEXTMACRO: FOO
  C" c:\apps\vfxforth" SETMACRO FOO

: BAR           \ --
  C" h:\myapp" SETMACRO FOO
;

$100 buffer: temp

<source> <dest> $EXPAND \ expand source string into destination
```

18.2 Basic words

`char % value textmac-delimiter        \ -- char ; text macro delimiter`

A `VALUE` returning the character used as the delimiter for macro names during text macro expansion. Unless you have a very good reason, do not change this value. In a future VFX Forth release, this word will be removed.

`: substituteC   \ src slen dest dlen --`

Expand the source string using text macro substitutions, placing the result as a counted string at *dest/dlen*. If an error occurred, the length of the counted string is zero.

: substituteZ \ src slen dest dlen --

Expand the source string using text macro substitutions, placing the result as a zero terminated string at *dest/dlen*. If an error occurred, the length of the string is zero.

: replaces \ text tlen name nlen --

Define the string text/tlen as the text to substitute for the substitution named name/nlen. If the substitution does not exist it is created.

: subsitute-safe \ c-addr1 len1 c-addr2 len2 -- c-addr2 len3 ior

Replace each '%' character in the input string *c-addr1/len1* by two '%' characters. The output buffer is represented by *caddr2/len2*. The output is *caddr2/len3* and ior is zero on success. If you pass a string through SUBSITUTE-SAFE and then SUBSTITUTE, you get the original string.

18.3 Utilities

: Expand \ caddr len -- caddr' len'

Macro expand the given string, returning a global buffer containing the expanded string. The string is zero-terminated and has a count byte before *caddr'*. If *len* is longer than 254 bytes only the first 254 bytes will be processed.

: MacroExists? \ caddr -- xt nz | 0

If a macro of the given name exists, return its xt and a non-zero flag, otherwise just return zero. The name is a counted string.

: MacroSet? \ caddr -- flag

If a macro of the given name exists and text has been set for it, return true. Often used to find out if a macro has been set, so that a sensible default can be defined. In the following example, IDIR is the current include directory and MC" is a version of C" that expands macros.

```
c" GuiLib" MacroSet? 0= [if]
  mc" %IDIR%" SetMacro GuiLib
[then]
```

: TextMacro: \ <"name"> --

Builds a new text-macro with an empty macro string.

```
TextMacro: Foo
```

: setmacro \ string "<name>" --

Reset/Create a text macro. Used in the form:

```
C" abcd" SETMACRO <name>
```

: $setmacro \ string name --

This version of SETMACRO takes both the string and macro name as counted strings.

: getTextMacro \ caddr len -- macro$

Given a macro name, return the address of its text (a counted string). If the name cannot be found the null counted string cNull is returned.

: .macros \ --

Display all text macros by macro name.

: .macro \ "<name>" -- ; .MACRO <name>

Display the text for macro <name>.

```
: ShowMacros    \ --
```
Display all macro names and text.

```
: $expand       \ $source $dest --
```
Macro expand a counted string at *$source* to a counted string at *$dest*.

```
: $ExpandMacros \ $ -- $'
```
Macro expand a counted string. Note that the returned string buffer is in a global buffer.

```
: z$ExpandMacros        \ z$ -- 'z$
```
Macro expand a 0 terminated string. The returned string buffer is a global buffer.

```
: ExpandMacro   \ c-addr u buff -- 'buff 'len
```
Perform TextMacro expansion on a string in *c-addr u* with the result being placed as a counted string at *buff*. The address and length of the expanded string are returned. The string at *'buff* is also zero terminated.

```
: M",           \ "text" --
```
Compile the text string up to the closing quote into the dictionary as a counted string, expanding text macros as `M",` executes, usually at compile time. The end of the string is aligned.

```
: MS"           \ Comp: "<quote>" -- ; Run: -- c-addr u
```
Like `S"` but expands text macros. Text is taken up to the next double-quotes character. Text macros are expanded at compile time. The address and length of the string are returned. To expand macros at run time, use:

```
  s" <string>" expand
```

```
: MC"           \ Comp: "<quote>" -- ; Run: -- c-addr
```
Like `C"` but expands text macros. Text is taken up to the next double-quotes character. Text macros are expanded at compile time. At run-time the address of the counted string is returned. To expand macros at run time, use:

```
  c" <string>" $ExpandMacros
```

18.4 System Defined Macros

The following text macros are defined by the system and are always available. They are implemented as words in the `substitutions` vocabulary.

```
create DevPath          ( -- c$ )   0 c, $FF allot
```
The path containing the developer's application source. If selected by setting `BuildLevel` to -1, the contents of this macro will be prepended to source file names in the `SOURCEFILES` vocabulary.

```
create VfxPath          ( -- c$ )   0 c, $FF allot
```
The path containing the VFX Forth source code. For most users, this is the *Sources* folder of the VFX Forth installation. You must set this yourself. It is preserved in the INI file.

```
create BasePath         ( -- c$ )   0 c, $FF allot
```
The root folder of the VFX Forth installation. You must set this yourself. It is preserved in the INI file.

```
create LOCATE_PATH      ( -- c$ )   0 c, $FF allot
```
The name of the current/last file to be compiled. Used by `LOCATE` and friends.

```
create LOCATE_LINE      ( -- c$ )   0 c, $FF allot
```
The line number of the line in the current file being compiled. Used by `LOCATE` and friends.

```
: f locate_path ;
```

The name of the current/last file to be compiled. A synonym for `LOCATE_PATH`. Used by `LOCATE` and friends.

`: l locate_line ;`

The line number of the line in the current file being compiled. A synonym for `LOCATE_LINE`. Used by `LOCATE` and friends.

`create LIBRARYDIR ( -- c$ ) 0 c, $FF allot`

The pathname of the VFX Forth *Lib* directory.

`: lib LIBRARYDIR ;`

A synonym for `LIBRARYDIR` above, which returns the pathname of the VFX Forth *Lib* directory.

`create LOAD_PATH ( -- c$ ) 0 c, $FF allot`

The directory containg the running program's executable.

`: bin LOAD_PATH ;`

The directory containing the VFX Forth executables. A synonym for `LOAD_PATH`.

`: idir \ -- c$`

The current include directory. This string is '.' if no file is being `INCLUDE`d and allows a load file to be in the form below. The load file can then be referenced from any other directory.

```
\ include \dir1\dir2\dir3\loadfile.fth
                              \ in loadfile.fth
 include %idir%\file1.fth     \ file1 in dir3
 include %idir%\file2.fth     \ file2 in dir3
 ...
```

`create wd \ -- c$`

The working directory. Under Windows, DOS, Unices and OS X this is `"."`. **Do not** change this macro.

End of text macros. What follows are normally exposed words.

`LOAD_PATH constant Forth-Buff \ -- caddr`

Returns the address of a counted string holding the directory from which the application was loaded. This gives programs access to the `LOAD_PATH` macro.

18.5 Linux specifics

The code described here is specific to VFX Forth for Linux. Do not rely on any of the words documented here being present in any other VFX Forth implementation.

`: GetExeName \ caddr --`

Get the fully qualified name of the executable. It is saved as a counted string at *caddr*.

`: InitMacros \ --`

Initialises the directory macros. Run at start up.

18.6 Editor and LOCATE actions

`#256 buffer: editor$ \ -- addr`

A buffer holding the path and name of the preferred editor as a counted string, e.g.

```
/bin/vi
```

`0 value EditOnError? \ -- flag`

Set true to call the editor on an error.

```
: editor-is     \ "<editor-name>" --
```
Set your preferred editor, e.g.

```
editor-is /bin/vi
```

```
: .ed           \ --
```
Display the name of your preferred editor

```
#256 buffer: locate$    \ -- addr
```
A buffer holding the macro expansion required to edit a specific line of a file. This information is used by `LOCATE`. In the example below the macros *%f%* and *%l%* will be replaced by the file name and line number.

```
%f% -# %l%
```

```
: edit$         \ -- cstring
```
If the preferred editor has been set, return the program name, otherwise return the default editor string for *vi*.

```
: $edit         \ cstring --
```
Edit the file provided as a counted string.

```
: edit          \ "<filename>" --
```
Edit the file whose name follows in the input stream, e.g.

```
EDIT release.txt
```

```
: (EditOnError) \ -- ; run editor on error
```
Edit the file at an error, using the contents of the variables `'SOURCEFILE` and `LINE#`.

```
: SetLocate     \ --
```
Tells VFX Forth how your editor can be called to go a particular file and line. Use in the form

```
SetLocate  <rest of line>
```

where the text after `SetLocate` is used to define how parameters are passed to the editor, e.g. for Emacs, use

```
SetLocate +%l% "%f%" &
```

The rest of line following `SetLocate` is used as the editor configuration string. Within the editor configuration string '%f%' will be replaced by the file name and '%l%' will be replaced by the line number. If you use file names with spaces, you should put quotation marks around the `%f%` text macro. Finish the line with `" &"` to run the editor detached from VFX Forth.

19 VFX Code Generator

The VFX code generator is a black box that simply does its job. Some implementations may have switches for special cases.

19.1 Enabling the VFX optimiser

The optimiser can be enabled and disabled by the words `OPTIMISED` and `UNOPTIMISED`. The state of the optimiser can be detected by inspecting the variable `OPTIMISING`.

19.2 Binary inlining

Binary inlining consists of copying the binary code for a word inline without the final return instruction. This avoids the overhead of the call and return instructions. It is useful for very short coded instruction sequences. For high level definitions the source inliner usually gives better results.

The VFX code generator gives some control over the use of binary inlining, controlled by the word `INLINING ( n -- )`. When the code generator has completed a word, the length of the word is stored. When the word is to be compiled, its length is compared against the value passed to `INLINING`, and if the length is less than the system value, the word is compiled inline, with the procedure entry and exit code removed. This avoids pipeline stalls, and is very useful for short definitions.

By default four constants are available for inlining control, although any number will be accepted by `INLINING`.

```
NO INLINING              \ 0, binary inlining turned off
NORMAL INLINING          \ 12-16, ~10% increase in size
AGGRESSIVE INLINING      \ 255, useful when time critical
ABSURD INLINING          \ 4096, unlikely to be useful
```

You can use `INLINING` anywhere in the code outside a definition.

19.2.1 Colon definitions

Any word that uses words that affect the return stack such as `EXIT`, or takes items off the return stack that you didn't put there in the same word, will automatically be marked as not being able to be inlined.

Implementations that use absolute calls will disable inlining of any word that makes an absolute call.

Note that when words are inlined, the effects may not be as expected.

```
: A ... ;                \ inlined
: B ... A ... ;          \ A inlined, B can be inlined
: C ... B ... B ... ;    \ A, B inlined, C can be inlined
```

19.2.2 Code definitions

By default `CODE` definitions are not marked for inlining because the assembler cannot detect all cases which may upset the return stack. If you want to make a code definition available for binary inlining, follow it with the word `INLINE`.

```
CODE <name>
  ...
END-CODE InLine
```

19.3 VFX Optimiser Switches

Some instructions are only available on later CPUs. Note that CPU selection affects the assembler and the VFX code code generator and compile time, **not** the run time instruction usage of your application. If you select a higher CPU level than the application runs on, incorrect operation will occur. The default selection is for the Pentium 4 instruction set.

```
CPU=386     \ -- ; select base instruction set
CPU=PPro    \ -- ; Pentium Pro and above with CMOVcc
CPU=P4      \ -- ; Pentium 4 and above
```

Aspects of the VFX code generator are controllable by switches. In particular the inlining of the `DO ... LOOP` entry code and local variable entry code may be turned on and off to suit your particular coding style.

Note also that for large computationally intensive definitions, the `SMALLER` and `FASTER` pair of switches may actually give better performance using `SMALLER`. The impact of these switches varies considerably between CPU types and cache/memory architecture.

`#16 value /code-alignment       \ -- n`
The default code alignment used by `FASTER` below. Must be a power of two.

`: smaller       \ --`
Selects smaller code using the minimum of alignment.

`: faster        \ --`
Selects faster code using 16 byte alignment, which will increase the size of the dictionary headers.

`: +polite       \ -- ; suppresses some warnings`
Suppresses some warning messages which some users may feel are commenting on their code. In particular, if you define constants to enable and disable code without using conditional compilation, you can use `+POLITE` to disable the warnings about conditional branches against a constant. See also `-POLITE`.

`: -polite       \ -- ; enables some warnings`
Enables some warning messages which warn you if have used a phrase such as `"<literal> IF"`. See `+POLITE`.

`0 value MustLoad?       \ -- n`
Returns true if indirect accesses are loaded rather than delayed.

`: +MustLoad     \ --`
Forces indirect memory loads to be fetched into a register rather than delayed. For some applications (mostly calculations with array indexing) this can lead to a performance gain.

`: -MustLoad     \ --`
Permits indirect memory loads to be delayed. This is the default condition.

`: +short-branches       \ --`
Enables the VFX optimiser to produce short forward branches. If your code causes a branch limit to be exceeded, you can put `-SHORT-BRANCHES` and `+SHORT-BRANCHES` around the offending words. By default, short branch generation is off because it gives better perforance on modern CPUs.

`: -short-branches       \ --`
Prevents the VFX optimiser producing short forward branches. By default, short branch generation is off.

`: short-branches?       \ -- flag ; true for short branches`
Returns true if the optimiser will produce short forward branches.

`: [-short-branches      \ -- sys`
Disables short branch optimisation until the previous state is restored by `SHORT-BRANCHES]`.

`: [+short-branches      \ -- sys`
Enables short branch optimisation until the previous state is restored by `SHORT-BRANCHES]`.

`: short-branches]       \ sys --`
restores the short branch optimisation previously saved by `+/-SHORT-BRANCHES]`.

`: LoopAlignment \ n --`
Set loop starts, e.g. `BEGIN..XXX` and `DO..LOOP` to be aligned on an n-byte booundary, where n must be a power of two. This is useful to force the heads of loops onto a cache line boundary. The default is 8.

```
#16 LoopAlignment     \ set to 16 byte boundary
0   LoopAlignment     \ revert to lowest setting
```

`: +fastlvs      \ --`
Enables generation of inline local variable entry code. This is the default condition, and is strongly recommended.

`: -fastlvs      \ --`
Disables generation of inline local variable entry code.
Most modern x86 operating systems use task gates for interrupt handling, which permits some code generation to be better, especially for local variables.

`SafeOS? value SafeOS?   \ -- flag`
Returns true if the operating system can be assumed to be safe.

`: +SafeOS       \ --`
Assume a safe modern operating system.

`: -SafeOS       \ --`
Assume an old-fashioned or raw operating system.

19.4 Controlling and Analysing compiled code

These directives control the optimiser

`: optimising?   \ -- flag`
Returns true if the optimiser is enabled.

```
: optimised     \ -- ; turn optimisation on
```

Enables the optimiser.

```
: unoptimised   \ -- ; turn optimisation off
```

Disables the optimiser.
These directives are used to turn optimisation on and off around sections of code.

```
: [opt          \ -- i*x
```

Save the current state of optimisation at the start of an `[OPT ... OPT]` structure. You can make no assumptions about what the data stack contains.

```
: [-opt         \ -- i*x
```

Save the current state of optimisation at the start of an `[-OPT ... OPT]` structure and turn optimisation off.

```
: opt]          \ i*x --
```

Restore the state of optimisation at the end of an `[OPT ... OPT]` structure to what it was at the start.
The following directives are `IMMEDIATE` words that you can put inside your definitions to obtain an idea of how code is being compiled. `DIS <name>` will disassemble a word.

```
: []            \ --
```

Lay a NOP instruction as a marker, without flushing the optimiser.

```
: [o/f]         \ --
```

Flush the optimiser state, generating the canonical stack state again with TOS in the EBX register, and all other stack items in the deep (memory) stack.

```
: [o/s]         \ --
```

Show the state of the optimiser's working stack.

19.5 Hints and Tips

On i32/x86 Pentium-class CPUs the PUSH and POP instructions generated by `>R` and `R>` are slow, and the VFX code generator is quite conservative in optimising return stack manipulations as compared with data stack anipulations. Although the code below is convenient, safe and easy to write it is slow. The `rect.xxx` words are fields in a structure.

```
: Rect@   \ rect -- l t r b
\ Retrieve the values x y r b from the RECT[ structure at
\ the address given.
  >r
  r@ rect.Left @
  r@ rect.Top @
  r@ rect.right @
  r> rect.bottom @
;
```

The version below generates far better code when performance is important.

```
: Rect@   \ rect -- l t r b
\ Retrieve the values x y r b from the RECT[ structure at
\ the address given.
  dup rect.Left @ swap
  dup rect.Top @ swap
  dup rect.right @ swap
  rect.bottom @
;
```

Because of the limited number of registers, better code is usually generated by passing a pointer to a structure such as a rectangle rather than passing four items on the data stack. Use of words such as `Rect@` should be reseved for preparing parameters for a Windows API call.

19.6 VFX Forth v4.x

If you have written custom optimisers, the EAX register is no longer free for use, but must be requested like any other working register. `CODE` definitions require no changes.

19.7 Tokeniser

From VFX Forth v4.3, build 2825, the tokeniser replaces the previous source inliner. The change was made to improve ANS and Forth200x standards compliance, and to reduce issues with particularly "guru" code. To prevent breaking your existing code, the tokeniser uses the same word names for its control words.

The tokeniser keeps track of what is compiled for a word, and reruns the compilation of short definitions rather than copying the compiled code inline. This gives the VFX code generator many more opportunities to remove stack operations and produces smaller and faster code while encouraging users to write short definitions. That having been said, the relationship of code size with and without the tokeniser enabled is obscure at best.

Under some rare conditions, usually those requiring tinkering with internal structures of VFX Forth during compilation, it is necessary to have a level of control over the tokeniser. This section documents those words.

19.8 Tokeniser state

`: discard-sinline       \ --`

Stops the current definition from being handled by the tokeniser. This is usually required by a compilation word which generates inline data, and for which repetition of the word containing the inline data would generate large code with little speed advantage.

`#128 Value SinThreshold \ -- u`

If the binary size of a word is less than this value, it can be tokenised. **SUBJECT TO CHANGE**.

`: .Tokens       \ xt --`

Display the token stream for a word.

`: .Tokeniser    \ --`

Display tokeniser state

19.8.1 Tokeniser control

`FALSE Value Sin?        \ -- flag`

A VALUE which enables tokenising when set. Using SIN? enables you to determine the state of the tokeniser

```
false value sindoes?    \ -- flag
```

A VALUE which enables tokenising of DOES> clauses when set. Using this value enables you to determine the state of the tokeniser.

```
false value SinActive?  \ -- flag
```

Returns true when the tokeniser is active. It is used to inibit some immediate words which must not be rerun when the word they are in is tokenised.

```
: +sin          \ --
```

Enable tokenising of following definitions.

```
: -sin          \ --
```

Disable tokenising of following definitions.

```
: +sindoes      \ --
```

Enable tokenising of the run time portions of defining words. Many defining words produced with CREATE ... DOES> have short run time actions. The address returned by DOES> is a literal and provides many opportunities for both space and speed optimisation.

```
: -sindoes      \ --
```

Disable tokenising of the run time portions of defining words.

```
: [sin          \ -- i*x
```

[SIN and SIN] define a range of source code and must be used interpretively (outside colon definitions). [SIN saves the current tokeniser state and SIN] restores it. Often used in the form:

```
[SIN -SIN ... SIN]
```

```
: sin]          \ i*x --
```

See [SIN above.

```
: [-sin         \ -- i*x
```

[-SIN saves the current tokeniser state, and turns off the tokeniser. SIN] restores the saved tokeniser state. Used in the form:

```
[-SIN ... SIN]
```

```
: [+sin         \ -- i*x
```

[+SIN saves the current tokeniser state, and turns on the tokeniser. SIN] restores the saved tokeniser state. Used in the form:

```
[+SIN ... SIN]
```

```
: Sinlined?     \ xt -- flag
```

Return true if the word defined by xt can be compiled by the tokeniser.

```
: RemoveSin     \ xt --
```

Remove tokeniser information from a word. If the word has no tokeniser information it is unaffected.

```
: DoNotSin      \ --
```

If the last word with a dictionary header must not be tokenised, place DoNotSin after its definition, e.g.

```
: foo ... ; DoNotSin
```

```
: IMMEDIATE     \ --
```

Mark the last defined word as immediate. Immediate words will execute whenever encountered regardless of STATE. IMMEDIATE also disables tokenising of the last defined word. In practice, this is not a performance issue as IMMEDIATE words are executed at compile time.

```
: RemoveSINinRange      \ start end --
```

Remove all tokeniser information for definitions within the given range.

```
: RemoveAllSins \ --
```

Remove all tokeniser data in the system. RemoveAllSins is executed by the exit chain during BYE.

19.8.2 Gotchas

These gotchas are very rare conditions. They usually only appear when you write words that affect the semantics (meaning) of compilation. You can use [-sin ... sin] to drill down to the words that are causing problems.

```
[-sin
: foo ... ;
: poo ... ;
sin]
```

Immediate and defining words

The tokeniser hooks into the guts of COMPILE, and LITERAL. Compilation performed through these words is unaffected by the tokeniser.

Tokenising of IMMEDIATE words is disabled to reduce problems with "guru" code. In nearly all cases, these words are only executed at compile time, so there is minimal impact on application performance. If an immediate word causes compilation using COMPILE, and LITERAL, the tokeniser will detect this and generate tokens, e.g.

```
: z1 postpone dup postpone over ; immediate
: z2 z1 ;
' z2 .tokens
StartToken
DUP
OVER
End Token
```

In the majority of cases the tokeniser handles defining words quite adequately. In a few cases, such as defining new types of xVALUE, better code generation can be obtained by performing some calculation at compile time. Such defining words should set a compiler for their children.

To do this, use SET-COMPILER and INTERP> rather than DOES>. INTERP> indicates to the compiler that what follows is performed when the child is interpreted and that a compiler for the child has been defined. The following example is the kernel definition of VALUE.

```
: value         \ n -- ; ??? -- ???
  create
    ,  ['] valComp, set-compiler
  interp>
    valInterp
;
```

Note that the chidren of words using `INTERP>` are **not** immediate - they have separate interpretation and compilation actions. `SET-COMPILER ( xt -- )` above sets `valComp,` to be the compiler of the last word `CREATE`d. `SET-COMPILER` takes the *xt* of the word it is to compile so that information can be extracted from the word.

There are rare occasions on which you may want to add a compiler to a non-defining word. Rather than making the word immediate and state-smart, which can lead to problems, you can add the compiler yourself. This is especially desirable when the compiler uses carnal knowledge of VFX Forth rather than just `COMPILE,` and `LITERAL`, The example is taken from the VFX Forth kernel.

```
: DO    \ Run: n1|u1 n2|u2 -- ; R: -- loop-sys
  NoInterp  ;
comp: drop  s_do,  3  ;
```

Return stack modifiers

In nearly all cases, words that modify the return stack will be detected and these words will not be tokenised. However, in some cases words containing such words should **not** be tokenised because the flow of control has been modified. The first example below fails, but the second does not. Note that, according to the ANS and Forth200x standards, these words are non-standard because they make the assumption that, on entry to a word, the top item on return stack is the return address. The example below is taken from a third-party application ported to VFX Forth.

This example is correctly detected, but fails because the code also requires the word containing `LIST>` not to be tokenised.

```
: list> ( thread -- element )
  BEGIN  @ dup  WHILE  dup r@ execute  REPEAT
  drop r> drop ;
...
: .fonts fonts LIST> .font ;
```

The example above makes two assumptions, one about the return stack in the use of `R@` and `R>`, and another about how colon definitions begin in `EXECUTE`.

The solution is to disable the tokeniser when the word is compiled. The containing word is forced to be untokenised.

```
: (list>) ( thread -- element )
  BEGIN  @ dup  WHILE  dup r@ execute  REPEAT
  drop r> drop ;
: list> ( thread -- element )
  postpone (list>) discard-sinline ; immediate
...
: .fonts fonts LIST> .font ;
```

If you need to write words such as these, partitioning them as above, plus careful use of `:NONAME` to create the second part improves portability and maintainability.

Using :

If you build a new compiling word that uses colon, `:`, its children can themselves be tokenised. If your new word saves and restores data from the return stack indirectly, the tokeniser may not detect this, leading to obscure runtime or compilation errors. This situation can be avoided by adding `DISCARD-SINLINE` after the use of colon, e.g.

```
: MY:   \ --
  :  postpone save-state  discard-sinline
;

: MY;   \ --
  postpone restore-state  postpone ;
;
```

Code size

Some coding styles can lead to excessive expansion of code size by the tokeniser. Apart from turning the tokeniser off, you can try reducing the size set in the value `SinThreshold`. Note that the relationship between the compiled size of a word and its equivalent after token expansion in another word is often obscure.

19.9 Code/Data separation

From VFX Forth v4.3 onwards, code/data separation is turned on by default.

19.9.1 Problem and solution

CPUs from the Pentium 3 onwards have serious performance problems when data is close to code, leading to a wide variation in performance depending on data location. Measurements on the random number generator in the benchmark suite had a variation of 7:1.

The file *Sources\Kernel\386Com\OPTIMISE\P4opt.fth* (with Professional and Mission versions) contains code for data space management for these processors. Results show that performance is improved by a factor of 2.3 on *BENCHMRK.FTH* and that performance is now independent of location. There is no degradation of performance on other CPUs. The code generation switches are:

```
+IDATA        \ -- ; enable code/data separation
-IDATA        \ -- ; dsable code/data separation
```

Note that when enabled, phrases such as

```
VARIABLE <name> <size> ALLOT
```

will **not** give the expected result. This is discussed in more detail below.

The solution is to separate code and data. When the optimisation is enabled, data is held in `IDATA` chunks away from code. There is no change to `CREATE`, `ALLOT`, comma and friends, which still operate on normal dictionary areas. The notation is derived from cross compiler usage in embedded systems.

19.9.2 Defining words and data allocation

The following is a conventional definition of a character/byte array defined in the dictionary.

```
: cCARRAY       \ n -- ; i -- c-addr
  CREATE  ALLOT  DOES> + ;
```

The data space reserved by ALLOT is intermingled with code, leading to bad performance. The second implementation is for best performance with P4 CPUs. `IRESERVE ( n -- c-addr )` reserves an n-byte block in the `IDATA` area and returns its address. The children of `ICARRAY` are made immediate in order to emulate the effect of the source inliner on children of `CCARRAY`. The implementation below is is illustrative only. State-smart words (considered "evil" by some) can be be avoided using `set-compiler` and `interp>`.

```
: icarray       \ n -- ; i -- c-addr
  dup ireserve  dup rot erase      \ reserve IDATA space
  create  immediate                \ children are IMMEDIATE
    ,                              \ address in IDATA
  does>
    @  state @ if                  \ compiling
      postpone literal  postpone +
    else                           \ interpreting
      +
    endif
;
```

In order to make the array defining word `CARRAY` independent of whether P4 optimisation is enabled `CARRAY` simply selects which version to use.

```
: CARRAY        \ n -- ; i -- c-addr
  idata?
  if  icarray  else  ccarray  endif
;
```

19.9.3 Gotchas

When `+IDATA` is in use, standard defining words such as `VARIABLE` and `VALUE` will reserve space in the `IDATA` areas, but `ALLOT` still reserves space in the dictionary. Consequently code such as:

```
VARIABLE <name> <size> ALLOT
```

will break when `+IDATA` is active. Use:

```
<size> BUFFER: <name>
```

for all such allocations.

Words such as `>BODY` and `BODY>` will not work correctly on words whose data area is in an IDATA region.

19.9.4 Glossary

`variable iblock          \ -- addr`
Holds the address of the current IDATA block.

`variable iblock#         \ -- addr`
Holds the size of the current IDATA block.

`variable idp             \ -- addr`
Holds the current location in the current IDATA block.

`variable def-igap        \ -- addr`
Holds the minimum code/data gap size, by default 8 kbytes.

`variable def-iblock#     \ -- addr`
Holds the default IDATA block size, by default 64 kbytes.

`: bin-align     \ n --`
Force alignment to an N byte boundary where N is a power of two. The space stepped over is set to 0.

`: alignidef     \ --`
Align the dictionary to the IDATA default boundary.

`: inoroom?      \ n -- flag`
Retuns true if there is not enough room in the current IDATA block.

`: make-iblock   \ n --`
Make an IDATA block that is at least n bytes long. If n is less than the default size in `DEF-IBLOCK#` the block will be the default size.

`: ialign        \ --`
Step the IDATA block pointer to the next 4 byte boundary

`: ialign16      \ --`
Step the IDATA block pointer to the next 16 byte boundary

`: ireserve      \ n -- a-addr`
Reserve n bytes in the current IDATA block.

`0 value idata?          \ -- flag`
Returns true if data is reserved in the IDATA block.

`: +idata        \ --`

Force data to be reserved in IDATA blocks.

`: -idata          \ --`
Data is reserved conventionally in the normal dictionary space.

`: 2variable      \ -- ; -- addr`
If `IDATA?` is true data is reserved in an IDATA block, otherwise it is reserved in the dictionary.

`: variable       \ -- ; -- addr`
If `IDATA?` is true data is reserved in an IDATA block, otherwise it is reserved in the dictionary.

`: buffer:        \ size -- ; -- addr`
If `IDATA?` is true data is reserved in an IDATA block, otherwise it is reserved in the dictionary.

`: value          \ n -- ; -- n`
If `IDATA?` is true data is reserved in an IDATA block, otherwise it is reserved in the dictionary.

`: 2value                 \ n -- ; -- n`
If `IDATA?` is true data is reserved in an IDATA block, otherwise it is reserved in the dictionary.

`: CARRAY         \ n -- ; i -- c-addr`
Creates a byte array. When the child executes, the address of the i'th byte in the array is returned. The index is zero based. If `IDATA?` is true data is reserved in an IDATA block, otherwise it is reserved in the dictionary.

```
10 CARRAY MYCARRAY              \ create 10 byte array
  5 MYCARRAY .          \ display address of element 5
```

`: ARRAY          \ n -- ; i -- a-addr`
Creates a cell size array. When the child executes, the address of the i'th cell in the array is returned. The index is zero based. If `IDATA?` is true data is reserved in an IDATA block, otherwise it is reserved in the dictionary.

```
10 CARRAY MYARRAY               \ create 10 byte array
  6 MYARRAY .           \ display address of element 6
```

20 Functions in DLLs and shared libraries

20.1 Introduction

VFX Forth supports calling external API calls in dynamic link libraries (DLLs) for Windows and shared libraries in Linux and other Unix-derived operating systems. Various API libraries export functions in a variety of methods mostly transparent to programmers in languages such as C, Pascal and Fortran. Floating point data is supported for use with *Lib\Ndp387.fth*.

Before a library function can be used, the library itself must be declared, e.g.

```
LIBRARY: Kernel32.dll
```

Access to functions in a library is provided by the `EXTERN:` syntax which is similar to a C style function prototype, e.g.

```
EXTERN: int PASCAL SendMessage(
  HWND hwnd, DWORD mesg, WPARAM wparam, LPARAM lparam
);
```

This can be used to prototype the function **SendMessage** from the Microsoft Windows API, and produces a Forth word `SendMessage`.

```
  SendMessage  \ hwnd mesg wparam lparam -- int
```

For Linux and other Unices, the same notation is used. The default calling convention is nearly always applicable. The following example shows that definitions can occupy more than one line. It also indicates that some token separation may be necessary for pointers:

```
Library: libc.so.6

Extern: int execve(
  const char * path,
  char * const argv[],
  char * const envp[]
);
```

This produces a Forth word `execve`.

```
  execve       \ path argv envp -- int
```

The parser used to separate the tokens is not ideal. If you have problems with a definition, make sure that `*` tokens are white-space separated. Formal parameter names, e.g. *argv* above are ignored. Array indicators, *[]* above, are also ignored when part of the names.

The input types may be followed by a dummy name which is discarded. Everything on the source line after the closing ')' is discarded.

From VFX Forth v4.3 onwards, PASCAL is the default calling convention in the Windows version. The default for the Linux and OS X versions is "C". The default is always used unless overridden in the declaration.

20.2 Format

```
EXTERN: <return> [ <callconv> ] <name> '(' <arglist> ')' ';'

<return>    := { <type> [ '*' ] |  void }
<arg>       := { <type> [ '*' ] [ <name> ] }
<args>      := { [ <arg>, ]* <arg> }
<arglist>   := { <args> | void }       Note: "void, void" etc. is illegal.
<callconv>  := { PASCAL | WINAPI | STDCALL | "PASCAL" | "C" }
<name>      := <any Forth acceptable namestring>
<type>      := ... (see below, "void" is a valid type)
```

Note that during searches `<name>` is passed to the operating system exactly as it is written, i.e. case sensitive. The Forth name is case-insensitive.

As a standard Forth's string length for dictionary names is only guaranteed up to 31 characters for portable source code, very long API names can cause problems. Therefore the word `AliasedExtern:` allows separate specification of API and Forth names (see below). `AliasedExtern:` also solves problems when API functions only differ in case or their names conflict with existing Forth word names.

20.3 Calling Conventions

In the discussion **caller** refers to the Forth system (below the application layer and **callee** refers to a a function in a DLL or shared library. The `EXTERN:` mechanism supports three calling conventions.

- C-Language: `"C"`

  Caller tidies the stack-frame. The arguments (parameters) which are passed to the library are reordered. This convention can be specified by using `"C"` after the return type specifier and before the function name. For Linux and most Unix-derived operating systems, this is the default.
- Pascal language: `"PASCAL"`

  Callee removes arguments from the stack frame. This is invisible to the programmer at the application layer The arguments (parameters) which are passed to the library are not reordered. This convention is specified by `"PASCAL"` after the return type specifier and before the function name.
- Windows API: `WINAPI | PASCAL | STDCALL`

  In nearly all cases (but **not all**), calls to Windows API functions require C style argument reversal and the called function cleans up. Specify this convention with `PASCAL`, `WinAPI` or `StdCall` after the return type specifier and before the function name. For Windows, this is the default.

Unless otherwise specified, the Forth system's default convention is used. Under Windows this is `WINAPI` and under Linux and other Unices it is `"C"`.

20.4 Promotion and Demotion

The system generates code to either promote or demote non-CELL sized arguments and return results which can be either signed or unsigned. Although Forth is an un-typed language it must deal with libraries which do have typed calling conventions. In general the use of non-CELL arguments should be avoided but return results should be declared in Forth with the same size as the C or PASCAL convention documented.

20.5 Argument Reversal

The default calling convention for the host operating system is used. The right-most argument/parameter in the C-style prototype is on the top the Forth data stack. When calling an external function the parameters are reordered if required by the operating system; this is to enable the argument list to read left to right in Forth source as well as in the C-style operating system documentation.

Under certain conditions, the order can be reversed. See the words `"C"` and `"PASCAL"` which define the order for the operating system. See `L>R` and `R>L` which define the Forth stack order with respect to the arguments in the prototype.

20.6 C comments in declarations

Very rudimentary support for C comments in declarations is provided, but is good enough for the vast majority of declarations.

- Comments can be `// ...` or `/* ... */`,
- Comments must be at the end of the line,
- Comments are treated as extending to the end of the line,
- Comments must not contain the ')' character.

The example below is taken from a *SQLite* interface.

```
Extern: "C" int sqlite3_open16(
  const void * filename,  /* Database filename [UTF-16] */
  sqlite3 ** ppDb         /* OUT: SQLite db handle */
);
```

20.7 Controlling external references

`1 value ExternWarnings? \ -- n`

Set this true to get warning messages when an external reference is redefined.

`0 value ExternRedefs?   \ -- n`

If non-zero, redefinitions of existing imports are permitted. Zero is the default for VFX Forth so that redefinitions of existing imports are ignored.

`1 value LibRedefs?      \ -- n`

If non-zero, redefinitions of existing libraries are permitted. Non-zero is the default for VFX Forth so that redefinitions of existing libraries and OS X frameworks are permitted. When set to zero, redefinitions are silently ignored.

`1 value InExternals?    \ -- n`

Set this true if following import definitions are to be in the `EXTERNALS` vocabulary, false if they are to go into the wordlist specified in `CURRENT`. Non-Zero is the default for VFX Forth.

`: InExternals   \ --`

External imports are created in the `EXTERNALS` vocabulary.

`: InCurrent     \ --`

External imports are created in the wordlist specified by `CURRENT`.

20.8 Library Imports

In VFX Forth, libraries are held in the `EXTERNALS` vocabulary, which is part of the minimum search order. Other Forth systems may use the `CURRENT` wordlist.

For turnkey applications, initialisation, release and reload of required libraries is handled at start up.

`variable lib-link       \ -- addr`

Anchors the chain of dynamic/shared libraries.

`variable lib-mask       \ -- addr`

If non-zero, this value is used as the mode for `dlopen()` calls in Linux and OS X.

`struct /libstr  \ -- size`

The structure used by a `Library:` definition.

```
  int >liblink          \ link to previous library
  int >libaddr          \ library Id/handle/address, depends on O/S
  int >libmask          \ mask for dlopen()
  0 field >libname      \ zero terminated string of library name
end-struct
```

`struct /funcstr \ -- size`

The structure used by an imported function.

`: init-lib      \ libstr --`

Given the address of a library structure, load the library.

`: clear-lib     \ libstr --`

Unload the given library and zero its load address.

`: clear-libs    \ --`

Clear all library addresses.

`: init-libs     \ --`

Release and reload the required libraries.

`: find-libfunction      \ z-addr -- address|0`

Given a zero terminated function name, attempt to find the function somewhere within the already active libraries.

`: .Libs         \ --`

Display the list of declared libraries.

`: #BadLibs      \ -- u`
Return the number of declared libraries that have not yet been loaded.

`: .BadLibs         \ --`
Display a list of declared libraries that have not yet been loaded.

`: Library:      \ "<name>" -- ; -- loadaddr|0`
Register a new library by name. If `LibRedefs?` is set to zero, redefinitions are silently ignored. Use in the form:

```
LIBRARY: <name>
```

Executing `<name>` later will return its load address. This is useful when checking for libraries that may not be present. After definition, the library is the first one searched by import declarations.

`: topLib        \ libstr --`
Make the library structure the top/first in the library search order.

`: firstLib      \ "<name>" --`
Make the library first in the library search order. Use during interpretation in the form:

```
FirstLib <name>
```

to make the library first in the search order. This is useful when you know that there may be several functions of the same name in different libraries.

`: [firstLib]    \ "<name>" --`
Make the library first in the library search order. Use during compilation in the form:

```
[firstLib] <name>
```

to make the library first in the search order. This is useful when you know that there may be several functions of the same name in different libraries.

20.8.1 Mac OS X extensions

The phrase `Framework <name.framework>` creates two Forth words, one for the library access, the other to make that library top in the search order. For example:

```
framework Cocoa.framework
```

produces two words

```
Cocoa.framework/Cocoa
Cocoa.framework
```

The first word is the library definition itself, which behaves in the normal VFX Forth way, returning its load address or zero if not loaded. The second word forces the library to be top/first in the library search order. Thanks to Roelf Toxopeus.

As of OSX 10.7, `FRAMEWORK` (actually **dlopen()**) will search for frameworks in all the default Frameworks directories:

- /Library/Frameworks
- /System/Library/Frameworks
- ~/Library/Frameworks

```
: framework     \ --
```
Build the two framework words. See above for more details. If `LibRedefs?` is set to zero, redefinitions are silently ignored.

20.9 Function Imports

Function declarations in shared libraries are compiled into the `EXTERNALS` vocabulary. They form a single linked list. When a new function is declared, the list of previously declared libraries is scanned to find the function. If the function has already been declared, the new definition is ignored if `ExternRedefs?` is set to zero. Otherwise, the new definition overrides the old one as is usual in Forth.

In VFX Forth, `ExternRedefs?` is zero by default.

```
variable import-func-link       \ -- addr
```
Anchors the chain of imported functions in shared libraries.

```
: ExternLinked  \ c-addr u -- address|0
```
Given a string, attempt to find the named function in the already active libraries. Returns zero when the function is not found.

```
: init-imports  \ --
```
Initialise Import libraries. `INIT-IMPORTS` is called by the system cold chain.

```
defer preExtCall        \ --
```
Windows only. A hook provided for debugging and extending external calls without floating point parameters or return items. It is executed at the start of the external call before any parameter processing.

```
defer postExtCall       \ --
```
Windows only. A hook provided for debugging and extending external calls without floating point parameters or return items. It is executed at the end of the external call after return data processing.

```
defer preFPExtCall      \ --
```
Windows only. A hook provided for debugging and extending external calls with floating point parameters or return items. . It is executed at the start of the external call before any parameter processing.

```
defer postFPExtCall     \ --
```
Windows only. A hook provided for debugging and extending external calls with floating point parameters or return items. It is executed at the end of the external call after return data processing.

```
: InExternals   \ --
```
External imports are created in the `EXTERNALS` vocabulary.

```
: InCurrent     \ --
```
External imports are created in the wordlist specified by `CURRENT`.

```
: Extern:       \ "text" --
```
Declare an external API reference. See the syntax above. The Forth word has the same name as the function in the library, but the Forth word name is **not** case-sensitive. The length of the function's name may not be longer than a Forth word name.

```
: AliasedExtern:        \ "forthname" "text" --
```

Like EXTERN: but the declared external API reference is called by the explicitly specified forthname. The Forth word name follows and then the API name. Used to avoid name conflicts, e.g.

```
AliasedExtern: saccept int accept( HANDLE, void *, unsigned int *);
```

which references the Winsock accept function but gives it the Forth name SACCEPT. Note that here we use the fact that formal parameter names are optional.

```
: LocalExtern:  \ "forthname" "text" --
```

As AliasedExtern:, but the import is always built into the CURRENT wordlist.

```
: extern        \ "text" --
```

An alias for EXTERN:.

```
: ExternVar     \ "<name>" -- ; ExternVar <name>
```

Used in the form

```
  ExternVar <name>
```

to find a variable in a DLL or shared library. When executed, <name> returns its address.

```
: AliasedExternVar      \ "<forthname>" "<dllname>" --
```

Used in the form

```
  AliasedExternnVar <forthname> <varname>
```

to find a variable in a DLL or shared library. When executed, <forthname> returns its address.

```
: .Externs      \ -- ; display EXTERNs
```

Display a list of the external API calls.

```
: #BadExterns   \ -- u
```

Silently return the number of unresolved external API calls.

```
: .BadExterns   \ --
```

Display a list of any external API calls that have not been resolved.

```
: func-pointer  \ xt -- addr
```

Given the XT of a word defined by EXTERN: or friends, returns the address that contains the run-time address.

```
: func-loaded?  \ xt -- addr|0
```

Given the XT of a word defined by EXTERN: or friends, returns the address of the DLL function in the DLL, or 0 if the function has not been loaded/imported yet.

20.10 Pre-Defined parameter types

The types known by the system are all found in the vocabulary TYPES. You can add new ones at will. Each TYPE definition modifies one or more of the following VALUEs.)

argSIZE
: Size in bytes of data type.

argDEFSIGN
: Default sign of data type if no override is supplied.

argREQSIGN
: Sign OverRide. This and the previous use 0 = unsigned and 1 = signed.

`argISPOINTER`
1 if type is a pointer, 0 otherwise

Each `TYPES` definition can either set these flags directly or can be made up of existing types.

Note that you should explicitly specify a calling convention for every function defined.

20.10.1 Calling conventions

`: "C"          \ --`

Set Calling convention to "C" standard. Arguments are reversed, and the caller cleans up the stack.

`: "PASCAL"     \ --`

Set the calling convention to the "PASCAL" standard as used by Pascal compilers. Arguments are **not** reversed, and the called routine cleans up the stack. This is **not** the same as `PASCAL` below.

`: PASCAL       \ --`

Set the calling convention to the Windows PASCAL standard. Arguments are reversed in C style, but the called routine cleans up the stack. This is the standard Win32 API calling convention. N.B. There are exceptions! This convention is also called "stdcall" and "winapi" by Microsoft, and is commonly used by Fortran programs. This is **not** the same as `"PASCAL"` above.

`: WinApi       \ --`

A synonym for `PASCAL`.

`: StdCall      \ --`

A synonym for `PASCAL`.

`: VC++         \ --`

Defines the calling convention as being for a C++ member function which requires "this" in the ECX register. The function must be defined with an explicit this pointer (void * this). Because exported VC++ member functions can have either "C" or "PASCAL" styles, the this pointer must be positioned so that it is leftmost when reversed (C/WINAPI/StdCall style) or is rightmost when not reversed ("PASCAL" style). See also the later section on interfacing to C++ DLLs.

`: R>L   \ --`

By default, arguments are assumed to be on the Forth stack with the top item matching the rightmost argument in the declaration so that the Forth parameter order matches that in the C-style declaration. `R>L` reverses this.

`: L>R   \ --`

By default, arguments are assumed to be on the Forth stack with the top item matching the rightmost argument in the declaration so that the Forth parameter order matches that in the C-style declaration. `L>R` confirms this.

20.10.2 Basic Types

`: unsigned     \ --`

Request current parameter as being unsigned.

`: signed       \ --`

Request current parameter as being signed.

`: int          \ --`

Declare parameter as integer. This is a signed 32 bit quantity unless preceeded by `unsigned`.

`: char \ --`

Declare parameter as character. This is a signed 8 bit quantity unless preceeded by `unsigned`.

`: void \ --`

Declare parameter as void. A `VOID` parameter has no size. It is used to declare an empty parameter list, a null return type or is combined with `*` to indicate a generic pointer.

`: * \ --`

Mark current parameter as a pointer.

`: ** \ --`

Mark current parameter as a pointer.

`: *** \ --`

Mark current parameter as a pointer.

`: const ; \ --`

Marks next item as **constant** in C terminology. Ignored by VFX Forth.

`: int32 \ --`

A 32bit signed quantity.

`: int16 \ --`

A 16 bit signed quantity.

`: int8 \ --`

An 8 bit signed quantity.

`: uint32 \ --`

32bit unsigned quantity.

`: uint16 \ --`

16bit unsigned quantity.

`: uint8 \ --`

8bit unsigned quantity.

`: LongLong \ --`

A 64 bit signed or unsigned integer. At run-time, the argument is taken from the Forth data stack as a normal Forth double with the top item on the top of the data stack.

`: LONG int ;`

A 32 bit signed quantity.

`: SHORT \ --`

For most compilers a **short** is a 16 bit signed item, unless preceded by `unsigned`.

`: BYTE \ --`

An 8 bit unsigned quantity.

`: float \ --`

32 bit float.

`: double \ --`

64 bit float.

`: bool1 \ --`

One byte boolean.

```
: bool4          \ --
```
Four byte boolean.

```
: ...            \ --
```
The parameter list is of unknown size. This is an indicator for a C varargs call. Run-time support for this varies between operating system implementations of VFX Forth. Test, test, test.

20.10.3 Windows Types

The following parameter types are non "C" standard and are used by Windows in function declarations. They are all defined in terms of existing types.

```
: OSCALL         PASCAL  ;
```
Used for portable code to avoid three sets of declarations. For Windows, this is a synonym for `PASCAL` and under Linux and other Unices this is a synonym for `"C"`.

```
: DWORD          unsigned int  ;
```
32 bit unsigned quantity.

```
: WORD           unsigned int  2 to argSIZE  ;
```
16 bit unsigned quantity.

```
: HANDLE         void *  ;
```
HANDLEs under Windows are effectively pointers.

```
: HMENU          handle  ;
```
A Menu HANDLE.

```
: HDWP           handle  ;
```
A DEFERWINDOWPOS structure Handle.

```
: HWND           handle  ;
```
A Window Handle.

```
: HDC            handle  ;
```
A Device Context Handle.

```
: HPEN           handle  ;
```
A Pen Handle.

```
: HINSTANCE      handle  ;
```
An Instance Handle.

```
: HBITMAP        handle  ;
```
A Bitmap Handle.

```
: HACCEL         handle  ;
```
An Accelerator Table Handle.

```
: HBRUSH         handle  ;
```
A Brush Handle.

```
: HMODULE        handle  ;
```
A module handle.

```
: HENHMETAFILE   handle  ;
```
A Meta File Handle.

```
: HFONT          handle  ;
```

A Font Handle.

`: HRESULT       DWORD   ;`
A 32bit Error/Warning code as returned by various COM/OLE calls.

`: LPPOINT       void *  ;`
Pointer to a POINT structure.

`: LPACCEL       void *  ;`
Pointer to an ACCEL structure.

`: LPPAINTSTRUCT void *  ;`
Pointer to a PAINTSTRUCT structure.

`: LPSTR         void *  ;`
Pointer to a zero terminated string buffer which may be modified.

`: LPCTSTR       void *  ;`
Pointer to a zero terminated string constant.

`: LPCSTR        void *  ;`
Another string pointer.

`: LPTSTR        void *  ;`
Another string pointer.

`: LPDWORD       void *  ;`
Pointer to a 32 bit DWORD.

`: LPRECT        void *  ;`
Pointer to a RECT structure.

`: LPWNDPROC     void *  ;`
Pointer to a WindowProc function.

`: PLONG         long *  ;`
Pointer to a long (signed 32 bit).

`: ATOM          word  ;`
An identifier used to represent an atomic string in the OS table. See **RegisterClass()** in the Windows API for details.

`: WPARAM        dword  ;`
A parameter type which used to be 16 bit but under Win32 is an alias for DWORD.

`: LPARAM        dword  ;`
Used to mean LONG-PARAMETER (i.e. 32 bits, not 16 as under Win311) and is now effectively a DWORD.

`: UINT          dword  ;`
Windows type for unsigned INT.

`: BOOL          int  ;`
Windows Boolean type. 0 is false and non-zero is true.

`: LRESULT       int  ;`
Long-Result, under Win32 this is basically an integer.

`: colorref      DWORD  ;`
A packed encoding of a color made up of 8 bits RED, 8 bits GREEN, 8 bits BLUE and 8 bits ALPHA.

`: SOCKET dword ;`
Winsock socket reference.

`: __in ;`
Microsoft header annotation.

`: __inout ;`
Microsoft header annotation.

`: __out ;`
Microsoft header annotation.

`: __in_opt ;`
Microsoft header annotation.

`: __inout_opt ;`
Microsoft header annotation.

`: __out_opt ;`
Microsoft header annotation.

`: _in_ ;`
Microsoft header annotation.

`: _inout_ ;`
Microsoft header annotation.

`: _out_ ;`
Microsoft header annotation.

`: _in_opt_ ;`
Microsoft header annotation.

`: _out_opt_ ;`
Microsoft header annotation.

20.10.4 Linux Types

`: OSCALL "C" ;`
Used for portable code to avoid three sets of declarations. For Windows, this is a synonym for `PASCAL` and under Linux this is a synonym for `"C"`.

`: FILE uint32 ;`
Always use as `FILE * stream`.

`: DIR uint32 ;`
Always use as `DIR * stream`.

`: size_t uint32 ;`
Linux type for unsigned INT.

`: off_t uint32 ;`
Linux type for unsigned INT.

`: int32_t int32 ;`
Synonym for `int32`.

`: int16_t int16 ;`
Synonym for `int16`.

`: int8_t int8 ;`

Synonym for `int8`.

`: uint32_t uint32 ;`
Synonym for `uint32`.

`: uint16_t uint16 ;`
Synonym for `uint16`.

`: uint8_t uint8 ;`
Synonym for `uint8`.

`: time_t uint32 ;`
Number of seconds since midnight UTC of January 1, 1970.

`: clock_t uint32 ;`
Processor time in terms of CLOCKS_PER_SEC.

`: pid_t int32 ;`
Process ID.

`: uid_t uint32 ;`
User ID.

`: mode_t uint32 ;`
File mode.

20.10.5 Mac OS X Types

`: OSCALL "C" ;`
Used for portable code to avoid three sets of declarations. For Windows, this is a synonym for `PASCAL` and under OS X this is a synonym for `"C"`.

`: FILE uint32 ;`
Always use as `FILE * stream`.

`: DIR uint32 ;`
Always use as `DIR * stream`.

`: size_t uint32 ;`
Unix type for unsigned INT.

`: off_t uint32 ;`
Unix type for unsigned INT.

`: int32_t int32 ;`
Synonym for `int32`.

`: int16_t int16 ;`
Synonym for `int16`.

`: int8_t int8 ;`
Synonym for `int8`.

`: uint32_t uint32 ;`
Synonym for `uint32`.

`: uint16_t uint16 ;`
Synonym for `uint16`.

`: uint8_t uint8 ;`
Synonym for `uint8`.

```
: time_t        uint32  ;
```
Number of seconds since midnight UTC of January 1, 1970.

```
: clock_t       uint32  ;
```
Processor time in terms of CLOCKS_PER_SEC.

```
: pid_t         int32  ;
```
Process ID.

```
: uid_t         uint32  ;
```
User ID.

```
: mode_t        uint32  ;
```
File mode.

20.11 Compatibility words

These words are mainly for users converting code from other Forth systems.

This section provides shared library imports in the form:

```
  function: foo  ( a b c d -- x )
```

where the brackets **must** be space delimited.

```
: FUNCTION:     \ "<name>" "<parameter list>" --
```
Generate a reference to an external function. The Forth name is the same as the name of the external function. Use in the form:

```
 function: foo1 ( a b c d -- }
 function: foo2 ( a b c d -- e }
 function: foo3 ( a b c d -- el eh }
```

The returned value may be 0, 1 or 2 items corresponding to void, int/long and long long on most 32 bit systems.

```
: ASCALL:       \ "<synonym-name>" "<name>" "<parameter list>" --
```
Generate a reference to an external function. The Forth name is not the same as the name of the external function. Use in the form:

```
 ascall: forthname funcname ( a b c d -- e }
```

```
: GLOBAL:       \ "<name>" --
```
Generate a reference to an external variable. Use in the form:

```
 global: varname
```

20.12 Using the Windows hooks

The hooks `preExtCall` and `postExtCall` are `DEFER`red words into which you can plug actions that will be run before and after any external call. They are principally used:

- To save and restore the NDP state when using screen and printer drivers that do not obey all the Windows rules.
- To save and restore the NDP state and you want the NDP state preserved regardless of any consequences. Although this is safe, the system overhead is greater than that of preserving your floats in variables or locals as required.

- Installing error handlers that work with nested callbacks.

The hooks `preFPExtCall` and `postFPExtCall` are compiled into calls with floating point parameters or return values. They do not affect the NDP state.

The examples below illustrate both actions.

20.12.1 Deferred words and variables

`defer preExtCall        \ --`

Windows only. A hook provided for debugging and extending external calls. It is executed at the start of the external call before any parameter processing.

`defer postExtCall       \ --`

Windows only. A hook provided for debugging and extending external calls. It is executed at the end of the external call after return data processing.

`defer preFPExtCall      \ --`

Windows only. A hook provided for debugging and extending external calls with floating point parameters or return items. It is executed at the start of the external call before any parameter processing.

`defer postFPExtCall     \ --`

Windows only. A hook provided for debugging and extending external calls with floating point parameters or return items. It is executed at the end of the external call after return data processing.

`variable XcallSaveNDP?  \ -- addr`

Set true when imports must save and restore the NDP state. Windows only. From build 2069 onwards, the default behaviour for Windows includes saving and restoring the FPU state. This can be inhibited by clearing `XcallSaveNDP?` before execution.

`variable abort-code     \ -- addr`

Holds error code for higher level routines, especially `RECOVERY` below. Windows versions only.

`variable aborting?      \ -- addr`

Holds a flag to indicate whether error recovery should be performed by a calling routine.

`defer xcall-fault       \ -- ; handles errors in winprocs`

Used by application code in the `DEFER`red words `preExtCall` and `postExtCall` above to install user-defined actions.

20.12.2 Default versions

```
code PreExtern  \ -- ; R: -- sys
\ Clears the abort code and saves the NDP state if XcallSaveNDP?
\ is set.
  mov   dword ptr abort-code , # 0      \ no previous abort code
  cmp     [] XcallSaveNDP? , # 0        \ Win: required
  nz, if,
    pop     eax
    lea     esp, -/fsave [esp]
    fsave   0 [esp]
    push    eax
  endif,
  ret
```

```
end-code
assign preExtern to-do preExtCall

code PostExtern \ -- ; R: sys --
\ Restore the NDP state if XcallSaveNDP? is set and test the
\ abort code.
  cmp   [] XcallSaveNDP? , # 0          \ required
  nz, if,
    pop     eax
    frstor  0 [esp]
    lea     esp, /fsave [esp]
    push eax
  endif,
  \ Detecting faults in nested callbacks.
  cmp   dword ptr abort-code , # 0      \ test previous aborting code
  nz, if,
    call  [] ' xcall-fault 5 +          \ execute xcall-fault if set
  endif,
  ret
end-code
assign postExtern to-do postExtCall

code PreFPExtern        \ -- ; R: -- sys ; SFP006
\ Clears the abort code.
  mov   dword ptr abort-code , # 0      \ no previous abort code
  ret
end-code
assign preFPExtern to-do preFPExtCall

code PostFPExtern       \ -- ; R: sys -- ; SFP006
\ Test the abort code.
  cmp   dword ptr abort-code , # 0      \ test previous aborting code
  nz, if,
    call  [] ' xcall-fault 5 +          \ execute xcall-fault if set
  endif,
  ret
end-code
assign postFPExtern to-do postFPExtCall
```

```
: DefaultExterns        \ --
```

Set the default PRE and POST EXTERN handlers.

20.12.3 Protected EXTERNs

Protected EXTERNs allow VFX Forth to recover when a crash occurs inside a Windows call and the Forth registers have been corrupted. For example

```
  255 0 GetCurrentDirectory
```

will crash because an address of zero is invalid. Protected EXTERNs save the Forth registers befor making the call so that exception handlers can restore VFX Forth to a known state.

```
code PreProtExtern      \ -- ; R: -- sys
\ Clears the abort code and saves the NDP state if XcallSaveNDP?
\ is set.
  mov    edx, # XcallBuffer            \ where the saved data goes
  mov    eax, 0 [esp]                  \ return address
  sub    eax, # 6                      \ xt of EXTERN (call [] prexx)
  mov    0 scb.xt [edx], eax           \ save it
  lea    eax, 4 [esp]                  \ RSP on entry
  mov    0 scb.esp [edx], eax
  mov    0 scb.ebp [edx], ebp
  mov    0 scb.esi [edx], esi
  mov    0 scb.edi [edx], edi

  mov   dword ptr abort-code , # 0     \ no previous abort code
  cmp     [] XcallSaveNDP? , # 0       \ Win: required
  nz, if,
    pop     eax                        \ return address
    lea     esp, -/fsave [esp]
    fsave   0 [esp]
    push    eax
  endif,
  ret
end-code

code PostProtExtern     \ -- ; R: sys --
\ Restore the NDP state if XcallSaveNDP? is set and test the
\ abort code.
  mov   dword ptr XcallBuffer scb.xt , # 0  \ reset Extern in progress
  cmp   [] XcallSaveNDP? , # 0         \ required
  nz, if,
    pop     eax                        \ return address
    frstor  0 [esp]
    lea     esp, /fsave [esp]
    push eax
  endif,
  \ Detecting faults in nested callbacks.
  cmp   dword ptr abort-code , # 0     \ test previous aborting code
  nz, if,
    call  [] ' xcall-fault 5 +         \ execute xcall-fault if set
  endif,
  ret
end-code

code PreProtFPExtern    \ -- ; R: -- sys ; SFP006
\ Clears the abort code.
  mov    edx, # XcallBuffer            \ where the saved data goes
  mov    eax, 0 [esp]                  \ return address
  sub    eax, # 6                      \ xt of EXTERN (call [] prexx)
  mov    0 scb.xt [edx], eax           \ save it
  lea    eax, 4 [esp]                  \ RSP on entry
  mov    0 scb.esp [edx], eax
  mov    0 scb.ebp [edx], ebp
```

```
  mov    0 scb.esi [edx], esi
  mov    0 scb.edi [edx], edi

  mov   dword ptr abort-code , # 0      \ no previous abort code
  ret
end-code

code PostProtFPExtern   \ -- ; R: sys -- ; SFP006
\ Test the abort code.
  mov   dword ptr XcallBuffer scb.xt , # 0      \ reset Extern in progress
  cmp   dword ptr abort-code , # 0      \ test previous aborting code
  nz, if,
    call  [] ' xcall-fault 5 +          \ execute xcall-fault if set
  endif,
  ret
end-code
```

```
: ProtectedExterns      \ --
```

Set the protected PRE and POST EXTERN handlers.

20.13 Interfacing to C++ DLLs

20.13.1 Caveats

These notes were written after testing on Visual C++ v6.0. Don't blame us if the rules change!

20.13.2 Example code

The example code may be found in the directory EXAMPLES\VC++. Because of the inordinate amount of time we spent wandering around inside debuggers to get this far, we recommend that you adopt a cooperative and investigative attitude when requesting technical support on this topic.

20.13.3 Accessing constructors and destructors

Example code for accessing the constructor of class is provided in *TRYCPP.FTH* which accesses the class DllTest in *DLLTEST.CPP*.

Since C++ is supposed to provide a higher level of abstraction, apparently simple operations may generate reams of code. So it is with the equivalent of

```
  pClass = new SomeClass;
```

The actual code generated may/will be a call to a function new to generate an object structure (not a single cell) followed by passing the return value from new to the class constructor.

The class constructor (in C++ CDllTest::CDllTest()) is not normally exported from C++ without some extra characters being added to the name. For example, the reference to it in the example code is:

```
  extern: PASCAL void * ??0CDLLTest@@QAE@XZ( void );
```

This function is not directly callable because it has to be passed the result of the `new` operator. To solve this problem *DLLTest.dll* contains a helper function CallNew which is passed the address of the constructor for the class. This is redefined as NEW for normal use.

```
\ C++ Helpers
extern: PASCAL void * CallNew( void * );
extern: PASCAL void CallDelete( void * this);
```

```
\ CDLLTest class specific
extern: PASCAL void * ??0CDLLTest@@QAE@XZ( void );

0 value CDLLTest        \ -- class|0

: InitCDLLTest  \ -- ; initialise the CPP interface
  ['] ??0CDLLTest@@QAE@XZ func-loaded? -> CDLLTest
;

: New   \ class -- *obj|0
  CallNew
;

: Delete        \ *obj --
  CallDelete
;
```

The word INITCDLLTEST gets the address of the constructor for the class, and NEW then runs the CallNew function which executes the C++ new operator and calls the constructor. Unfortunately, you will have to do this for each class that use in the DLL. What is returned by CallNew is an object pointer. This is not the object itself, but the address of another (undocumented) data structure. It can be used as the this pointer for all following member function calls.

Once you have finished with the onject, you must relase its resources using the delete method (the destructor). This is implemented in VC++ by passing the object pointer to the delete function. This is performed by the CallDelete function exported from the DLL. Again, the Forth word DELETE provides syntactic sugar by just calling CallDelete.

20.13.4 Accessing member functions

A Visual C++ member function exported from a DLL requires the "this" pointer in the ECX register. This can be achieved using the following form:

```
extern: VC++ PASCAL BOOL TestWindow1( void * this, char * ch, int n, int nbyte );
```

The function must be defined with an explicit this pointer (void * this). Because exported VC++ member functions can have either C or "PASCAL" styles, the this pointer must be positioned so that it is leftmost when reversed (C/PASCAL/WINAPI/StdCall/APIENTRY style) or is rightmost when not reversed ("PASCAL" style).

```
extern: PASCAL VC++ BOOL GetHello( void * this, char * buff, int len );
extern: PASCAL VC++ BOOL TestWindow1( void * this, char * ch, int n, int nbyte );
extern: PASCAL VC++ BOOL TestWindow2( void * this, void * pvoid, int ndword, int nlong );

0 value CDLLTest        \ -- constructor/class

: InitCDLLTest  \ --;  Initialise the CPP interface
  ['] ??0CDLLTest@@QAE@XZ func-loaded? to CDLLTest
;

create Magic#  $AAAA5555 ,      \ -- addr

#64 buffer: StringBuff  \ -- ; buffer for GetHello

: TestCDLLTest  \ -- ; test CDLLTest interface
  InitCDLLTest  CDLLTest if
    cr ." Initialisation succeeded"
    CDLLTest new ?dup if
      cr ." new succeeded"
      dup StringBuff #64 GetHello drop
      cr ." GetHello returns: " StringBuff .z$
      dup Magic# 4 5 TestWindow1 drop
      dup Magic# #20 #30 TestWindow2 drop
      delete
      cr ." delete done"
    else
      cr ." new failed"
    endif
  else
    cr ." Initialisation failed"
  endif
;
```

Please note that the actual code in TRYCPP.FTH may/will be different as we extend the facilities. See the source code itself!

20.13.5 Accessing third party C++ DLLs

Most third party C++ DLLs are provided with C header files which define the interfaces. Study of these will provide the information you need to determine how to access them.

For simple C++ classes, the DllTest.dll file can be used to provide constructor and destructor access. Note that classes with multiple constructors will export these as functions with the same basic name differentiated by the name mangling.

The DLL *Fth2VC60.dll* contains new and delete access for use with other DLLs. Note that the third party DLLs must be compatible with VC++ v6.0. The example file *EXAMPLES\VC++\USECPP.FTH* demonstrates using *Fth2VC60.dll*.

```
library: Fth2VC60.dll

extern: PASCAL void * FTH2CPPNew( void * constructor);
extern: PASCAL void FTH2CPPDelete( void * this);

: New           \ *class -- *obj|0
  FTH2CPPNew
;

: Delete        \ *obj --
  FTH2CPPDelete
;
```

If you are using an incompatible compiler or DLL, create a similar support DLL for that compiler. You can use the source code for *Fth2VC60.dll* as an example.

20.14 Changes at v4.3

The guts of the `EXTERN:` mechanism have been rewritten to provide more features and to support more operating systems.

20.14.1 Additional C types

The following C data types are now supported:

- Float - a 32 bit floating point item.
- Double - a 64 bit floating point item
- LongLong - a 64 bit integer. These are taken from and returned to the Forth data stack as Forth double numbers with the high portion topmost on the stack.

Floating point numbers are taken from the NDP floating point unit. This is directly compatible with the the Forth floating point pack in *Lib\Ndp387.fth*.

20.14.2 More Operating Systems

The requirements of newer operating systems, especially those for 64-bit operation, are more stringent for things like data alignment. Consequently the underlying mechanism has changed.

20.14.3 Miscellaneous

These notes are probably only relevant for code that has carnal knowledge of the VFX Forth internals.

- The `XCALL` primitive has been removed and is replaced by `NXCALL`.
- The compile time code generation is completely different and there is no centralised despatch mechanism.
- Some faciliies provided by *Lib\Win32\SafeCallback.fth* are now built in to the Windows system. You **must** use the new version of *SafeCallback.fth*.

The word `NXCALL` is provided for constructing your own import mechanisms, but it only deals with single-cell arguments and provides no type safety at all. It is used internally by VFX Forth in the first stage build of a console-mode kernel.

```
code NXCALL     \ i*x addr i -- res
```

Calls the operating system function at *addr* with *i* arguments, returning the result *res*. As far as the operating systems is concerned, *i*x* appear on the CPU return stack pointed to by ESP, and the return value is taken from EAX. After executing `NXCALL` the return value *res* is the contents of the EAX register.

21 C Language Style Helpers

VFX Forth supports a few "helper" definitions to aid the parsing of "C" header files.

These are especially helpful in the parsing of Windows resource scripts which are based on the Microsoft RC Language for C, and for cutting and pasting from C header files.

```
: #define       \ <spaces"NAME"> <eol"value-def"> -- ; Exec: -- value
```

A simple version of C's #define preprocessor command. Any text between the definition name and the end of the line is EVALUATEd when <NAME> is invoked.

```
: //            \ --
```

An implementation of the C++ single line comment.

```
: /*            \ --
```

A simple implementation of the C "/* ... */" comment.

```
: enum          \ --
```

Process an enum of the form:

```
  enum <name> { a, b, c=10, d };
```

<name> is ignored. The elements appear as Forth constants. The definition may extend over many lines. C comments may occur after the ',' separator, e.g.

```
 JIM = 25, // comment about this line
```

```
: enum{         \ --
```

Process an enum of the form:

```
  enum{ a, b, c=10, d };
```

22 Supported shared libraries

This chapter documents interfaces to shared libraries that can be used on all VFX Forth versions with the `Extern:` shared library interface - all except DOS. We will add more library interfaces as time goes by. Supported libraries can be found in *<Vfx>/Lib/SharedLibs/*.

We will support the interface code here, but our technical support cannot include teaching you how to use these libraries. Generous we may be, a charity we are not.

Open Source libraries can be found using Google, and may already be installed on your machine. For example, the SQLite database engine is used by FireFox and many other projects.

A reliable source of binaries for Windows is the MinGW distribution from:

`http://sourceforge.net/projects/mingw/`

`http://www.mingw.org/`

We will periodically put the Windows versions (and perhaps others) of the libraries on our FTP site at:

`ftp://soton.mpeforth.com/`

You can use most browsers to access this. Login as "public" with a blank password and switch to the *SharedLibs* directory.

22.1 LibCurl

The *libcurl* library/DLL provides high-level functions for transferring data across networks to and from servers. be found at:

```
http://curl.haxx.se/libcurl/
```

Additional help may be found at the curl-library mailing list subscription and unsubscription web interface:

```
http://cool.haxx.se/mailman/listinfo/curl-library/
```

`struct /curl_httppost   \ -- len`

The Forth version of the curl_httppost structure.

`0x10000001 constant CURL_WRITEFUNC_PAUSE`

This is a magic return code for the write callback that, when returned, will signal libcurl to pause receiving on the current transfer.
Note that all Curl callbacks must be defined with `FromC`.

`struct /curl_fileinfo   \ -- len`

Content of this structure depends on information which is known and is achievable (e.g. by FTP LIST parsing). Please see the url_easy_setopt(3) man page for callbacks returning this structure – some fields are mandatory, some others are optional. The FLAG field has special meaning.

22.2 LibIconv

LibIconv converts from one character encoding to another through Unicode conversion (see Web page for full list of supported encodings). It has also limited support for transliteration, i.e. when a character cannot be represented in the target character set, it is approximated through one or several similar looking characters. It is useful if your application needs to support multiple character encodings, but that support lacks from your system.

The latest version of the library can be obtained from

`http://gnuwin32.sourceforge.net/packages/libiconv.htm`

The required files are:

- libiconv2.dll
- libcharset1.dll
- libintl3.dll
- a compatible *msvcrt.dll*
- iconv.exe
- libiconvman.pdf - the library documentation
- libiconv.fth

To obtain a full list of the supported encodings, go to a operating system command line and type:

```
  iconv -l
```

`: iconv_t  void *  ;`

Define the `iconv_t` type as is done by the C header file.

`: size_t        uint32  ;`

Type for unsigned INT.

`Extern: iconv_t "C" libiconv_open( const char * tocode, const char * fromcode );`

Allocates descriptor for code conversion from encoding *fromcode* to encoding *tocode*.

`Extern: size_t "C" libiconv(`

Converts, using conversion descriptor *cd*, at most **inbytesleft* bytes starting at **inbuf*, writing at most **outbytesleft* bytes starting at **outbuf*.

Decrements **inbytesleft* and increments **inbuf* by the same amount.

Decrements **outbytesleft* and increments **outbuf* by the same amount.

```
  iconv_t cd,
  const char * * inbuf, size_t * inbytesleft,
  char * * outbuf, size_t * outbytesleft
);
```

`Extern: int "C" libiconv_close( iconv_t cd );`

Frees resources allocated for conversion descriptor *cd*.

22.3 SQLite

The most recent version of SQLite can be found at:

```
http://www.sqlite.org/
```

This is a very compact and fast Open Source SQL database system that is ideal for managing data in a single application.

The VFX Forth interface can be found in the direcory:

```
<Vfx>/Lib/SharedLibs/SQLite3
```

`: pFunc  void *  ;`
Pointer to a function.

`: sqlite3  void  ;`
This should always appear as `sqlite3 *`. Since this is an opaque type, translating it to `void *` is valid.

`: sqlite3_stmt  void *  ;`
Essentially a string pointer.

`: sqlite3_value void *  ;`
Can be anything!

`: sqlite3_context void *  ;`
Can hold anything!

`: sqlite3_blob void *  ;`
Pointer to an object.

`: sqlite3_vfs  void *  ;`
Essentially a pointer.

`: sqlite3_mutex  void *  ;`
Essentially a pointer.

22.4 zlib

The *zlib* library/DLL provides high-level functions for compression and decompression. The current version of the library is at

```
www.zlib.net
```

The file *zlib.fth* is a conversion of *zlib.h* for VFX Forth. The *zlib* version is 1.2.5. If you are using this code with a different version of zlib, use

```
zlibCompileflags ( -- x )
```

to check that the types and fields `zLong` and `zInt` defined here are correct.

22.4.1 Windows specifics

The most reliable source of binaries for Windows is the MinGW distribution from:

```
http://sourceforge.net/projects/mingw/
http://www.mingw.org/
```

```
[defined] Target_386_Windows [if]
library: zlib1.dll
also types definitions
: zexport   "C"  ;
: zlong     uint32  ;   \ uLong type
: zInt      uint32  ;   \ uInt type
: z_off_t   uint32  ;
previous definitions
: zlong     4 field  ;
: zInt      4 field  ;
[then]
```

22.4.2 Mac OS X specifics

```
[defined] Target_386_OSX [if]
\ library: libcurl.2.dylib
also types definitions
: zexport   "C"  ;
: z_off_t  LongLong  ;
: zlong     uint32  ;   \ uLong type
: zInt      uint32  ;   \ uInt type
previous definitions
: zlong     4 field  ;
: zInt      4 field  ;
[then]
```

22.4.3 Linux specifics

```
[defined] Target_386_Linux [if]
\ library: libcurl.so.3
\ library: libcurl.so.4
also types definitions
: zexport   "C"  ;
: z_off_t  LongLong  ;
: zlong     uint32  ;   \ uLong type
: zInt      uint32  ;   \ uInt type
previous definitions
: zlong     4 field  ;
: zInt      4 field  ;
[then]
```

22.4.4 Generic code

```
struct /z_stream_s      \ -- len
```

z_stream_s structure.

22.5 LibXL - Excel interface

LibXL is a library that can read and write Excel files. It doesn't require Microsoft Excel or the .NET framework, and combines a set of easy to use and powerful features. The library can be used to:

- Generate a new spreadsheet from scratch
- Extract data from an existing spreadsheet
- Edit an existing spreadsheet

LibXL is proprietary code, but the price is very reasonable. Versions exist for Windows, OS X, iOS and Linux. *LibXL* can be obtained from:

```
www.libxl.com
```

This code is built as 32 bit code using the ASCII interface. Consequently, strings shown with **L"** in the C examples are ASCII zero-terminated strings and the Forth word `z"` can be used for them. The DLL interface uses the "C" calling convention for all operating systems. The Windows version should use *libXL.dll* from the *bin* directory of *LibXL* distribution.

`: enum          \ --`

Process an enum of the form:

```
enum <name> { a, b, c=10, d };
```

<name> is ignored. The elements appear as Forth constants. The definition may extend over many lines. C comments may occur after the ',' separator, e.g.

```
JIM = 25, // comment about this line
```

`: enum{         \ --`

Process an enum of the form:

```
enum{ a, b, c=10, d };
```

22.5.1 Test code

`: UniPlace      \ addr len destaddr --`

Store a Unicode string to an address. The string is stored as a cell counted string with a 16 bit zero terminator. The terminator is not included in the count.

`: UniAppend ( addr len destaddr -- )`

Append a string to the end of an address

`: Ascii>Uni,    \ addr len --`

Store an ASCII string as a Unicode string in the dictionary. The string is stored as a cell counted string with a 16 bit zero terminator. The terminator is not included in the count.

`: Ascii>Uni     \ addr len dest --`

Store an ASCII string as a Unicode string at an address. The string is stored as a cell counted string with a 16 bit zero terminator. The terminator is not included in the count.

`: UniCount ( addr -- addr len )`

Fetch a unicode string from an address.

`: LZ"           \ "text" -- zaddr`

Unicode string - should behave the same way as z"

`: xlTest        \ --`

libXL example 1

23 Callback functions

The CALLBACK mechanism provides the facility to wrap Forth definitions in code which is callable by Linux. The Forth stacks and data areas are created as frames on the calling stack.

23.1 Simple CALLBACK functions

`variable ip-default     \ -- addr`

Holds the default value of `IP-HANDLE` that is set for each CALLBACK entry.

`variable op-default     \ -- addr`

Holds the default value of `OP-HANDLE` that is set for each CALLBACK entry.

`: set-callback  \ xt callback --`

Make the xt be the action of the callback.

`: callback,     \ #in #out -- address`

Lay down a callback data structure. The first cell contains the address of the entry point. The address of the data structure is returned.

`: CALLBACK:     \ #in #out "<name>" -- ; -- a-addr`

Create a callback function. *#IN* and *#OUT* refer to the number of input and output parameters required for the callback. When the definition `<name>`is executed it will return the address of the callback function. For example

```
  2 1 CallBack: Foo
```

creates a callback named `Foo` with two inputs, and one output. Executing `Foo` returns the entry point used by Linux. To use it, pass `Foo` as the entry point required by Linux, e.g as the address of a task action. `Foo` is built to use the `"C"` calling convention.

```
  ' FooAction to-callback foo
```

Having defined an action for the callback, you can now use the callback as if it was a C or assembler function called by the operating system.

`: CallProc:     \ #in #out "<name>" -- ; -- entry`

Create a callback function and start compilation of its action. *#IN* and *#OUT* refer to the number of input and output parameters required for the callback. When the definition `<name>`is executed it will return the entry point address of the callback function.

```
4 1 CallProc: <name>  \ #in #out -- ; -- entry
\ Callback action ; x1 x2 x3 x4 -- op
  ...
;
<name>   \ returns entry point address
```

`: to-callback   \ xt <"name"> --`

Assign an XT as the action of a defined callback. This word is state smart.

23.2 An example. Creating a signal handler

A Linux signal handler has the prototype

```
void sa_siginfo( int signum, siginfo_t * siginfo, ucontext_t * uc );
```

As far as Forth is concerned we need to execute a Forth word that receives three parameters and returns none.

```
(SigTrap)   \ signum *siginfo *ucontext --
```

The code fragment below achives this.

```
3 0 callback: SigTrap  \ -- addr
\ executing SigTrap in Forth returns the C entry point.

: (SigTrap)  \ signum *siginfo *ucontext --
\ Action of SigTrap.
  nip                                  \ discard siginfo
  cr
  cr ." Signal number " swap .sigName
  uc.mcontext                          \ point at CPU context
  cr ." at address " dup sc.EIP @ dup .dword
     ." , probably in " ip>nfa .name
  cr
  ['] SigThrow swap sc.EIP !           \ force return to SigThrow
;
assign (SigTrap) to-callback SigTrap
```

The callback entry code provides you with a default I/O device and sets `BASE` to decimal. It does **not** set up a default `THROW` handler. If your callbacks must cope with exceptions, you must provide a top-level `CATCH` yourself.

23.3 Implementation notes

Callbacks are (usually) C functions. In the case of VFX Forth these functions create a Forth environment with two or more stacks, a `USER` area and so on. In a GUI environment, callbacks are very common, and so must be established and discarded quickly. The easiest place to do this is to use the calling C stack and build the Forth stacks and data areas on the C stack. This has several consequences:

- VFX Forth uses stacks generously, so the C stack has to be large. Between 1 and 4Mb is common. The need for this amount of memory is caused by a callback triggering another callback. We have observed nesting levels of 12 or more in applications.
- To improve performance, VFX Forth does not "touch" the stack it needs when the Forth environment is created. Therefore all the stack space must be "committed" in advance.
- Only a limited number of `USER` variables are initialised: `S0`, `R0`, `BASE`, `IP-HANDLE`, `OP-HANDLE`, `ThreadExit?`, `ThreadTCB`, and `ThreadSync`.
- You can make **no** assumptions about the addresses used for the Forth environment. Two separate invocations of the same callback may use quite different addresses. You can assume that the addresses will not change within a callback unless the application has changed them.
- Callbacks, tasks and signals all use the same entry mechanism.

24 Building Standalone Programs

24.1 The basics

After all the initialisation has been performed, the deferred word ENTRYPOINT is executed. The most basic way to make a turnkey application is just to set ENTRYPOINT and then SAVE the application. Some examples follow.

Later sections in this chapter discuss startup and shutdown in detail.

24.1.1 Windows GUI

You can use a messagebox or a modal dialog as the application; anything that runs the Windows message pump will work.

```
: start  \ hmodule 0 commandline show -- res
  4drop
  0 z" Hello World" z" VFX Forth" MB_OK MessageBox drop
  0
;
' start is EntryPoint
Save hello
bye
```

24.1.2 Windows console

```
: start  \ hmodule 0 commandline show -- res
  4drop
  ." Hello World! from VFX" cr
  0
;
' start is EntryPoint
SaveConsole hello
bye
```

24.1.3 OS X and Linux console

```
: start  \ hmodule 0 commandline show -- res
  4drop
  ." Hello World! from VFX" cr
  0
;
' start is EntryPoint
SaveConsole hello
bye
```

24.2 Sequence of Events

When a Forth program runs there are five stages from program launch to termination.

`OS_Startup`
: The code required to bridge the gap between Forth and the host operating system. This code is handwritten by MPE and cannot be changed in any manner (to do so would cripple the system)

`Initialisation`
: The setup of various system variables, stack pointers, user areas etc. Also the initialisation of import DLL functions etc. The various words which perform these operations are formed into a linked list called the Cold Chain which is executed automatically at start up.

`BootStrap`
: The execution of either the default interpreter or the end-user's turnkey application. There is a little stub of code which sets up some input parameters before calling the `DEFER`ed word `EntryPoint`.

`EntryPoint`
: This is the application code which runs until the user or the program decides it is time to shut down.

`Shutdown`
: When the application terminates, VFX Forth runs an exit chain similar to the cold chain. Any actions required for a clean shutdown should be added to this chain.

To generate stand-alone executable programs, three steps are required.

- Compile your application - you do this in the same way as you would during development. The required initialisation and shutdown actions are usually defined using

  `xt AtCold`

  `' foo AtExit`
- Assign your entry point - you must define a word which serves as your program entry point and assign it as the action of the entry point word `EntryPoint`.
- Save the compiled image - after compilation and entry point definition you commit your compiled code to a file.

24.3 The `EntryPoint` word

At the end of the start up chain `EntryPoint` is called. This word is `DEFER`ed and can be re-assigned by the user.

The entrypoint definition is supplied by the end-user for a turnkey application. The system has a default entry point which simply launches the interpreter. An entry point definition has the following format:-

```
MyEntryPoint    \ hmodule 0 commandline show -- res
```

Where the parameters are:

`HMODULE`
: The "module handle". For instance under Windows the module-handle is the base address of the parent process when running in memory.

`0`
: A reserved field.

`COMMANDLINE`
: A zero terminated string from the operating system describing the command line used to launch the system. Where this information is unavailable this field will be 0.

`SHOW`
: An operating system specific field describing what visual effect should be used to start the application. In Windows this value can be passed directly to **ShowWindow()**.

`RES`
: The result with which to exit the program.

The syntax used to reset the entry point is:

```
  ASSIGN <myword> TO-DO ENTRYPOINT
or
  ' <myword> IS ENTRYPOINT
```

and should be placed at the end of your source build before `SAVE`.

Under some operating systems and I/O devices, you must flush pending output before shutting down, otherwise it will be unseen (still be buffered) when the program terminates. For example, this may not work.

```
: start  \ hmodule 0 commandline show -- res
  4drop
  ." Hello World! from VFX" cr
  0
;
Assign start to-do EntryPoint
SaveConsole hello
bye
```

What happens is that the buffered output has not been displayed before the program terminates. To fix this there are three options:

- Use `flushOP-gen drop` to flush the current output.
- Use `key? drop` as I/O drivers (should) flush output when an input request is made.
- Use `200 ms` if you are uncertain or the O/S layer needs time, as can happen in some networking situations.

Note that under some operating systems you cannot save a file of the same name as the one that is currently executing.

24.4 Startup and Shutdown words

```
: ShowColdChain          \ --
```

Show on the console the sequence of events which make up the current start up code. The sequence is shown in the order in which it is executed.

```
: ShowExitChain          \ --
```

Show on the console the sequence of events which make up the current exit actions. The sequence is shown in the order in which it is executed.

`: WalkColdChain          \ --`

Walk the cold chain. Used during startup.

`: WalkExitChain          \ --`

Walk the exit chain. Used during shutdown.

`: AtCold         \ xt --`

Specify a new XT to execute when the Cold chain sequence is run.

`: AtExit         \ xt --`

Add a new XT to execute on BYE.

`: FREEZE         \ --`

Setup initial user area/global values for `SAVE`. `FREEZE` is performed by the guts of `SAVE`.

`: (init)         \ --`

Set up system variables, task0 user area, search order etc.

`variable ExitCode        \ -- addr`

Holds exit code returned to the operating system.

`: bye            \ --`

Runs the shutdown chain, and exits to the operating system, returning the exit code from the variable `ExitCode`. The meaning of the exit code is operating system dependent.

`: cold           \ --`

System entry point. Runs the cold chain and then the application `EntryPoint` code. When the application finishes, the exit chain is run and the application terminates.

24.5 Saving to an ELF file

The following sequence performs this operation:

```
ok SAVE myfile.elf
ok BYE
poop> ./myfile.elf
```

`: Mb             \ n -- nMb ; nMb = n * 1048576`

Given n, returns n megabytes. Useful before `SET-SIZE` or `ALLOCATE`.

`: Kb             \ n -- nKb ; nKb = n * 1024`

Given n, returns n kilobytes. Useful before `SET-SIZE` or `ALLOCATE`.

`: get-size       \ -- size`

Return the amount of memory used by the Forth dictionary and system headers.

`: set-size       \ size --`

Set the amount of memory to be used by the Forth dictionary and system headers. This will not take effect until the system has been `SAVE`d to form a new ELF file. The new dictionary size will be used when the new ELF file is run.

`: get-stacks     \ -- size`

Return the amount of memory used by the Forth stacks and user area.

`: set-stacks     \ size --`

Set the amount of memory to be used by the Forth stacks and user area. This will not take effect until the system has been `SAVE`d to form a new ELF file. The new size will be used when the new ELF file is run.

`: Save            \ "<name>" -- ; SAVE <name>`

Save the application to the file given by the following file name. The .ELF extension must be supplied. The image is saved as a complete ELF file.

25 Exception and Error Handling

25.1 CATCH and THROW

`CATCH` and `THROW` form the basis of all VFX 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`.

25.1.1 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 VFX Forth implementation is similar to the one described above, but is not identical.

25.1.2 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
;
```

25.1.3 Wordset

`: >ep           \ x --`

Push a cell item onto the exception stack.

`: ep>           \ -- x`

Pop a cell item from the exception stack.

`defer o_ABORT   \ i*x -- ; R: j*x --`

The exception handler of last resort. Clears the stacks and calls the `DEFER`red word `QUIT`.

`: CATCH         \ i*x xt -- j*x 0|i*x n                          9.6.1.0875`

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                             9.6.1.2275`

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. If n is non-zero and no previous `CATCH` has been performed, there is an exception frame error, and `O_ABORT` is performed which finally calls the `DEFER`red word `QUIT`.

`: ?THROW        \ k*x flag throw-code -- k*x|i*x n`

Perform a `THROW` of value *throw-code* if flag is non-zero.

25.1.4 Extending CATCH and THROW

The `CATCH` and `THROW` mechanism can be extended by the user if additional information needs to be preserved. By default the following pointers are preserved - data stack, return stack, float stack, local frame and current object. Additional information is saved and restored by user-defined words as follows.

The save word must return n, the number of cells saved on the exception stack. The restore word must consume n and restore n items from the return stack. The words `>EP ( x -- )` and `EP> ( -- x )` are used to push and pop items respectively to and from the exception stack.

```
variable v1
variable v2

: MySave     \ -- n ; number of cells
  v1 @ >ep  v2 @ >ep  2
;

: MyRestore  \ n -- ; number of cells
  drop  ep> v2 !  ep> v1 !
;
```

The new save and restore words are activated by the word `EXTENDS-CATCH` and the default action is restored by `DEFAULT-CATCH`. See below.

`: extends-catch \ xt-save xt-restore --`
Sets the new save and restore actions of `CATCH` and `THROW`.

`: default-catch \ --`
Restores the default actions of `CATCH` and `THROW`.

25.2 ABORT and ABORT"

These words are built on top of `CATCH` and `THROW`.

`defer ABORT     \ i*x -- ; R: j*x -- ; error handler`
Empty the data stack and perform the action of `QUIT`, which includes emptying the return stack, without displaying a message.

`: ABORT"        \ Comp: "ccc<quote>" -- ; Run: i*x x1 -- | i*x ; R: j*x -- | j*x  9.6.2.0680`
If x1 is true at run-time, display the following string and perform `ABORT`, otherwise do nothing. This is handled by performing `-2 THROW` after setting the variable `'ABORTTEXT`.

25.3 Defining Error/Throw codes

As of VFX Forth v3.6, the user definable mechanism has changed.

In order to simplify the construction and allocation of error messages and references, they can be constructed automatically. Use `ERRDEF` and `#ERRDEF` as shown below to construct messages for error handlers. These messages are created as `/ERRDEF` structures which are also used for messages that can be internationalised. Note also that this structure and mechanism may be subject to change to cope with internationalisation, which is documented in a separate chapter of the manual.

```
ErrDef ScrewUp "Oh bother, something went wrong"
```

defines a constant called `SCREWUP` associated with a string. The constant `SCREWUP` can be passed to ERR$ to retrieve the address and length of the string. The value of the constant is generated from the contents of variable NEXTERROR.

```
999 #ErrDef Snafu "Situation Normal, All ****ed Up"
```

defines a constant called SNAFU of value 999 and an associated text message.

When assigning error codes, please note that the ANS specification reserves error codes -255..-1 for ANS defined error messages. Error codes in the range -4095..-256 are reserved for use by VFX Forth iteself. Applications may use other codes. Please do not use error codes 0..499 as these are reserved by VFX Forth for optional system extensions. By default, automatically assigned error codes start at 501.

The error system relies on a data structure /ERRDEF which follows a constant value for the error number. The /ERRDEF structure contains a link to the previous ERRDEF or #ERRDEF definition, a message identifier which is 0 for non-databased strings in the ISO Latin1 coding, the address of the text, and the length of the text in bytes. The text is followed by two zero bytes, and the text is long aligned. The /ERRDEF structure is a subset of the /TEXTDEF structure described in the internationalisation chapter. This chapter also includes a discussion of the concepts used for internationalisation.)

Error messages are linked into the chain used for all application strings that can be internationalised. This chain is anchored by the variable TEXTCHAIN.

The words PARSEERRDEF, ERR$ and .ERR are DEFERred. PARSEERRDEF creates the /ERRDEF structure from the source text. It is the basis of error defining words such as ERRDEF. You can install alternative versions of these words for internationalised applications. In this context, #ERRDEF and friends can be used as the basis of any text handler that requires translation. Note that PARSEERRDEF can be modified so that a message file is produced at compile time, and ERR$ modified so that the message file is accessed at run time. Similarly, providing that the application language is correctly handled, the run time can access translated messages in other languages, character sets and character sizes. .ERR is similarly DEFERred and is used to display the message.)

```
struct /errdef  \ -- len ; DOES NOT include constant definition
  int ed.link           \ link to previous ERRDEF
  int ed.id             \ 0 or message ID
  int ed.caddr          \ address of text string to use
  int ed.len            \ length of text string to use in bytes
  int ed.lenInline              \ length of inline text string in bytes
end-struct
```

The previous kernel words GETERRORTEXT and GETERRORTEXTEX that existed up to VFX Forth v3.3 have been removed and are replaced by ERR$, which has the same stack effect.

`defer ParseErrDef       \ "<text>" -- ; create /ErrDef structure`

Create a /ErrDef structure in the dictionary, parsing the required text. The error number must already have been laid.

`defer Err$      \ n -- addr/n len/0`

Convert an error/message number to the address of the relevant string. *Addr* is the start of the string, and *len* is its length in bytes. If the string cannot be found, *addr* is set to *n* and *len* is set to 0.

```
defer .Err      \ caddr u --
```

Given the address and length in bytes of a message that may have been internationalised, display it. The default action is TYPE.

```
variable NextError
```

Holds the value of the next application error number to be allocated by ERRDEF. Application error numbers are positive and are incremented by ERRDEF.

```
variable NextSysError
```

Holds the value of the next system error number to be allocated by SYSERRDEF. System error numbers are negative and are decremented by SYSERRDEF.

```
variable TextChain
```

The anchor for the chain of error and text messages that may be internationalised.

```
: ErrStruct     \ n -- struct|0 ; produce pointer to error structure
```

Given an error number n, return the address of the /ERRDEF error structure containing its details.

```
: .TextChain    \ -- ; display all error messages
```

Display a list of all the error codes and messages defined by ERRDEF and #ERRDEF and other text chain users.

```
: (Err$)        \ throw#/msg# -- c-addr u | throw#/msg# 0
```

The default action of ERR$.

```
: (ParseErrDef) \ "<text>" -- ; create /ErrDef structure
```

The default action of PARSEERRDEF.

```
: #ErrDef       \ n -- ; -- n ; used as throw/error codes
```

Define a constant and associated message in the form:

```
  <n> #ERRDEF <name> "<text>".
```

Execution of <name> returns <n>.

```
: ErrDef        \ -- ; -- n ; used as throw/error codes
```

Define a constant and associated message in the form:

```
  ERRDEF <name> "<text>"
```

Execution of <name> returns the constant automatically allocated from NEXTERROR.

```
: SysErrDef     \ -- ; -- n ; used as throw/error codes
```

Define a constant and associated message in the form:

```
  SYSERRDEF <name> "<text>"
```

```
: #AnonErr      \ n "<text>" -- ; create anonymous error text
```

Create an anonymous error definition that has no name, e.g.

```
WSAEWOULDBLOCK #AnonErr "WSAEWOULDBLOCK"
```

#AnonErr is useful when dealing with operating system return codes whose names are available from the support DLL.

25.4 System Error Handling

The VFX Forth kernel handles the display of error messages using the word `.THROW ( n -- )` which recognises two classes of message. The first class consists of the messages handled by `ABORT" <text>"`. At present these cannot be internationalised, and they are displayed by the deferred word `DOABORTMESSAGE`. The second class handles all other error messages, and these are displayed by the deferred word `DOERRORMESSAGE`.

See also:

`LINE#` Current source input line number.

`'SourceFile`
: Pointer to source include struct for current source file, or 0.

`SOURCE` Return source line ; – c-addr u

`CurrSourceName`
: Return source file name ; – c-addr u

Other subsidiary words are also documented here.

```
: .ErrDef        \ n --
```
Display the error/message number n and the associated message.

```
: ShowErrorLine \ -- ; display error line
```
Show the current source file and line number. If `LINE#` contains -1, no action is taken.

```
: ShowSourceOnError     \ -- ; display pointer to error
```
Show the current text input line and a pointer to the error location defined by the current value of `>IN`. If `>IN` contains -1, no action is taken.

```
: .source-line  \ --
```
Show the current source file, line and a pointer to the source position.

```
defer DoAbortMessage    \ c-addr --
```
The action of this word is used by the kernel to print the text associated with an `ABORT"` from the system. By default it simply `TYPE`s the counted string at c-addr, and shows the offending source line. You can replace this action at any time, using:

```
ASSIGN <xxx> to-do DoAbortMessage
```

Note that `ABORT"` messages cannot be internationalised at present because all these messages share a common throw code, -2. `DOABORTMESSAGE` is used by `.THROW` below.

```
defer DoErrorMessage    \ n --
```
A deferred word which handles the display of error messages for the VFX Forth system's text interpreter. By default the `ERRDEF` mechanism above is used. `DOERRORMESSAGE` is used by `.THROW` below.

```
: .throw        \ n -- ; show throw code n
```
Process the given `THROW` code. Throw codes 0 and -1 are silent by specification. Throw code -2 displays the string set by the last `ABORT" <string>"` and all other error messages are searched for in the `ERRDEF` chain. See `ERRDEF` and `#ERRDEF`.

```
create zSysName \ -- zaddr
```
Returns the system name `"VFX Forth"` as a zero terminated string.

26 Using libgtk2 widgets from Forth

The source code for his chapter can be found in the directory *Examples/GTK*. Additional directories provide tutorial code, including a small editor.

GTK provides a GUI interface and graphical toolkit for many operating systems. It also includes a GUI designer program called *Glade* whose output can be read and used by GTK applications. This chapter is not a tutorial on GTK programming.

Several GTK+ tutorials exist, including

```
http://developer.gnome.org/gtk-tutorial/
```

The book we used during development is *Foundations of GTK+ Development*, by Andrew Krause, published by APress, ISBN 978-1-59059-793-4.

The file *gtkbindings.fth* links to the libgtk library and allows GTK calls to be made from Forth. This provides VFX Forth with a powerful widget library that works on Windows, Linux and Mac OS X.

The file *gtktools.fth* provides basic functions including the main GTK message loop.

The file *graphics.fth* provides a graphics interface in the style of the old Borland BGI interface. This is still very useful for many applications.

The file *GraphicsDemo.bld* loads all needed files and provides a number of demonstrations, of which `ShowGraphics3` is the most useful.

The file *TextEd/editor.fth* contains a simple text editor based on a tutorial by Micah Carrick.

26.1 External Linkages

The code loads required libraries and defines necessary API calls. There are two points to be noted when using this code:

- The DLLs/libraries must be in the library search order when VFX Forth runs or compiles this code.
- Under Windows, we use the MinGW version of GTK+. Like much Open Source software built with gcc, it uses a C calling convention both for external functions and for callbacks.

```
: CCB:          \ #in #out "<name>" -- ; -- entry
```

Behaves as `CallBack:` but enforces a C parameter-passing convention.

```
: CCBproc:      \ #in #out "<name>" -- ; -- entry
```

Behaves as `CallProc:` but enforces a C parameter-passing convention.

26.2 Signal Connection

26.2.1 Recommended

```
extern: long "C" g_signal_connect_object(
  void * instance, char * detailed_signal, void * c_handler,
  void * gobject, uint connect_flags
);
extern: long "C" g_signal_connect_data(
  void * instance, char * detailed_signal, void * c_handler, void * data,
  int destroy_data, uint connect_flags
);

: g_signal_connect      \ instance detailed_signal c_handler data -- ulong
  0 0 g_signal_connect_data
;
```

26.2.2 Macros

The GTK function **gtk_signal_connect_full()** is the library signal connection function; the C header file defines the other connect-style functions as macros. These words create the interface as it is more usually used.

```
extern: unsigned int "c" gtk_signal_connect_full(
  int * object,
  char * name,
  int * func,
  int * unsupported,
  int * data,
  void * destroy_func,
  int object_signal,
  int after
);
```

```
: gtk_signal_connect \ object name func data --
```

Connect an event handler to a signal.

```
: gtk_signal_connect_after \ object name func data --
```

Connect an event handler to a signal, but at the end of the event chain.

```
: gtk_signal_connect_object   \ object name func slot_object
```

Connect an event handler which takes a *GtkObject* pointer as its argument to a signal.

```
: gtk_signal_connect_object_after   \ object name func slot_object
```

Connect an event handler which takes a *GtkObject* pointer as its argument to a signal, at the end of the event chain.

26.3 GTK Types

GTK operates on a number of different object types - and these are frequently checked on call (with assertion failures if the wrong type is used). These words should cause the object to be cast correctly.

```
\ object -- cast_object
GTK_OBJECT
GTK_CONTAINER
GTK_BOX
GTK_WIDGET
GTK_WINDOW
```

`struct /GError \ -- len`
Definition of a **GError** structure.

26.4 GTK message pump

`: gtk-step \ -- flag`
Iterate GTK's inner loop, process any events and exit immediately. *flag* is returned true if a GTK event was processed.

`: gtk-step-blocking \ -- flag`
Iterate gtk's inner loop, process any events, wait until there are events if none are waiting when the function was called.

`: gtkPump \ --`
Empty the GTK event queue.

26.4.1 Windows

`: gui-Idle \ --`
Wait for message or GTK events and process it/them.

`: gui-BusyIdle \ --`
If a message or GTK events are available, process them.

`: gui-EmptyIdle \ --`
While messages and GTK events are available, process them.

`: gui-AppIdle \ --`
Process messages until the **GetMessage** API call returns zero in response to a WM_QUIT message. The VFX Forth variable `ExitCode` is set to the **wParam** value of this message. – This version includes the GTK inner loop. –

`: installGTKhooks \ --`
Install the GTK+ versions of the message pumps.

26.4.2 Linux

`: gui-WaitIdle \ --`
Wait for message or GTK events and process it/them.

`: gui-BusyIdle \ --`
If a message or GTK events are available, process them.

`: gui-EmptyIdle \ --`
While messages and GTK events are available, process them.

`: installGTKhooks \ --`
Install the GTK+ versions of the message pumps.

26.4.3 Mac OS X

`: gui-WaitIdle \ --`
Wait for message or GTK events and process it/them.

`: gui-BusyIdle  \ --`

If a message or GTK events are available, process them.

`: gui-EmptyIdle \ --`

While messages and GTK events are available, process them.

`: installGTKhooks       \ --`

Install the GTK+ versions of the message pumps.

26.5 Operating System Dependencies

26.5.1 Linux

If GTK+ and Glade are not already installed, install them using the system's package manager.

26.5.2 Windows

Under Windows, we use the MinGW version of GTK+. Like much Open Source software built with gcc, it uses a C calling convention both for external functions and for callbacks.

GTK+

We get our GTK + binaries from:

```
http://www.gtk.org/download/win32.php
```

The Windows version of GTK+ used by MPE is supplied as a directory tree that you place on your PC. During development, the simplest way to work is to add the *bin* folder to your PC's system path. Depending on your version of Windows, this is done from Control Panel -> System -> Advanced -> Environment Variables. You may have to reboot after changing the path.

If your application is being distributed into environments controlled by an IT department, it may not be possible to alter the system path. In this case make sure that the GTK libraries and executables are in the same folder as your application.

Glade

Glade is the GU designer for GTK+. At first, we tried to install Glade to use the version of GTK+ used for application run-time. This is a mistake! Installing a separate matched Glade/GTK+ pair permits much more reliable development. MPE gets Glade and GTK+ as one package from:

```
http://ftp.gnome.org/pub/GNOME/binaries/win32/glade3/
```

If you are feeling adventurous, go to:

```
http://glade.gnome.org/
```

26.5.3 Mac OS X

We use the GTK-OSX package. You can either use the package built by MPE and supplied as *gtkbins.tgz* or you can work from the source package. At application run time, it is usually sufficient to use the prebuilt binaries if you use absolute paths to the GTK shared libraries. For development, install from the source package. One day, we'll find a suitable incantation to run *Glade* from the binary package.

Installing gtkbins.tgz

Launch a terminal, select your home folder and unpack the package:

```
tar -zxvf gtkbins.tgz
```

Executables will then be in *~/gtkbins/bin* and shared libraries in *~/gtkbins/lib*. Edit *gtkbindings.fth* and other files such as *gladetest.fth* so that library references are correct.

Installing from source

During devlopment, it is convenient to use the source distribution, although it is tedious to install.

If you want to use the *Glade* designer and you already have MacPorts or Fink installed, you will need to create a new user and install GTK+ for that user in order to avoid conflicts.

Instructions for installing it are at:

`http://sourceforge.net/apps/trac/gtk-osx/wiki/Build`

Read these instructions fully and carefully before you start. In particular read the *Snow Leopard* section and apply the changes for a 32 bit build **before** running *jhbuild*. Then perform the three main GTK+ builds:

```
$ jhbuild bootstrap
$ jhbuild build meta-gtk-osx-bootstrap
$ jhbuild build meta-gtk-osx-core
```

Then build the *libglade* library needed at runtime and the *Glade* designer needed for development.

```
jhbuild build libglade
jhbuild build glade3
```

Executables will then be in *~/gtk/inst/bin* and shared libraries in *~/gtk/inst/lib*. Edit *gtkbindings.fth* and other files such as *gladetest.fth* so that library references are correct.

In order to run *Glade* properly, it should be launched by

```
$ jhbuild shell
$ glade-3 &
```

This incantation is required to persuade *Glade* to find its icons.

26.6 Loading GTK Builder files

```
: zFindCallback \ zCbName -- entry true | 0
```

Search the default search order for the given callback name. On success, the entry point is returned.

```
7 0 CCBproc: BuilderConnect_cb  \ *Builder *object zSignal zHandler *ConnObj flags *User --
```

The main Glade signal connection callback. This is called for each signal handler specified in the Glade file, i.e. handlers named by the GUI designer. Handler names are looked up in the current context. The words found are linked to the appropriate object by way of the GTK signal connection bindings.

```
variable CurrBuilder    \ -- addr
```

Holds the current builder handle.

```
: loadBuilderXML        \ z$ -- builder|0
```

Load a Glade GtkBuilder XML file and return the pointer or zero on error. On success, the connections are made and the variable `CurrBuilder` is set.

```
: BuilderObject \ z$ -- *widget
```

Obtain the widget pointer for the given named widget from the current builder object.

```
: freeBuilder   \ --
```

Free the current builder.

26.7 Dialogs

```
: runDlg        \ dialog -- response
```

Given a ***GtkDialog**, run the dialog as a modal dialog and return the result.

```
: ErrorBox      \ zMessage zTitle parent --
```

Displays an error box with the given message, title and parent window.

```
: AskSaveFileNameBox    \ zaddr len parent -- flag
```

Given a buffer and parent window, ask for a file name. On success, the filename is returned (clipped as required) in the buffer as a zero terminated string. If no file name is given the buffer is left unchanged.

```
: AskOpenFileNameBox    \ zaddr len parent -- flag
```

Given a buffer and parent window, ask for an existing file name. On success, the filename is returned (clipped as required) in the buffer as a zero terminated string. If no file name is given the buffer is left unchanged.

26.8 Event Callbacks

The GTK library calls back into Forth on various events. Here are some sample definitions. You may need to modify them or add new ones according to the complexity of your UI. You can also use this code to test your GTK installation.

Compile the file *gtkbindings.fth*. Then run

```
 .libs  .badExterns
```

This will display the load addresses of the libraries and show you any unresolved external functions. A library value of zero indicates that the library has not been found. If everything is good at this point, run the test window:

```
  gtktest
```

which should display a "hello world" window.

```
: delete_event_fn \ *widget *event *data -- 0/1
```

Handles delete events. It returns zero so that the widget is destroyed.

```
3 1 CCB: delete_event \ -- addr ; *widget *event *data -- 0/1
```

Callback for the delete event. Return 0 to perform the default handling or return 1 to indicate that the callback has done everything necessary.

```
: destroy_fn         \ *widget data --
```

Handles the destroy event for the application. It calls **gtk_main_quit()**.

```
2 0 CCB: destroy_event
```

Callback for the destroy event.

```
: wd_destroy    \ addr --
```

Destroy the widget

```
1 0 CCB: widget_destroy_cb      \ -- entry
```

Callback to handle the destroy event.

26.9 GTK startup and shutdown

```
0 value GTKstarted?     \ -- x
```

Non-zero when GTK has been started.

```
0 value AppFinished?    \ -- x
```

Non-zero when app must quit.

```
0 value gtk_main?       \ -- x
```

Non-zero if `gtk_main` is in use.

```
: noVfxGtk      \ --
```

Mark the VFX Forth to GTK interface as unused.

```
: do_gtk_init   { | temp[ cell ] -- }
```

Run **gtk_init**.

```
: do_gtk_main   \ --
```

The tools version of `gtk_main`.

```
: initGTK       \ --
```

Call this to initialise the GTK system. Always call this word to start the GTK system.

```
: GtkAppQuit    \ --
```

The GTK+ app should finish.

26.10 GTK test code

```
: hello_world_window    \ --
```

Launch the "hello world" window.

```
: gtktest       \ --
```

Start GTK+, launch the hello world window, and wait until it closes.

```
: hw            \ --
```

As `gtktest` but does not initialise GTK+ again.

26.11 Graphics in the Borland style

26.11.1 Global Data

`struct /gwindow \ -- len`
Structure to control a graphics window.

`variable windows        \ -- addr`
Anchors chain for keeping track of all created windows.

`CELL +USER CANVAS       \ -- addr`
The current drawing window.

26.11.2 Internal operations

`: window>       \ -- window`
Get the current drawing window.

`: penx          \ -- addr`
The current window's X coordinate for subsequent drawing commands.

`: peny          \ -- addr`
The current window's Y coordinate for subsequent drawing commands.

`: filled?       \ -- addr`
The current window's internal flag affecting drawing commands; see `filled`.

`CREATE mycolor   0xffff0000 , 0xffff w, 0 w, 0 w,`
Initial foreground color for new windows

`: window-dims   \ window -- w h`
Get the dimensions of a window from the window structure.

`: load-pixbuf   \ zaddr -- pixbuf`
Load an image into a Gtk_Pixbuf, which can be drawn to the current window using `PUT`.

`: redraw        { window -- }`
Make changes to a window's graphics visible.

`: ?redraw   { window -- }`
Internal - redraw a window only if it is "dirty" (affected by any drawing commands). Immediately resets the window's dirty flag, making it "clean" again

`: ChainEach \ ... xt anchor -- ...`
Execute xt on the contents of chain with the following structure:

```
  link | ...
```

The given xt must have the stack effect

```
  ... link -- ...
```

Where link is the address of the link field in the structure.

You can pass your own values to each link, just remember to clean up afterwards.

`1 1 CCBproc: timeout_event_cb   \ 0 -- true ; -- entry`
Callback run from a timer to redraw all "dirty" windows.

`: +gtimer       \ --`

Start the graphics event timer.

`: -gtimer       \ --`
Stop the graphics event timer

`: winResized    \ window --`
Perform this when the window has been resized.

`3 0 CCBProc: GframeCallback     \ window event data -- ; -- entry`
Callback to force window size to be updated and the window redrawn.

`3 0 CCBProc: GExposeCallback    \ window event data -- ; -- entry`
Callback to force window to be redrawn.

`: filled>       \ -- flag`
Fetch the filled flag

`: drawdest>     \ -- pixmap gc`
Fetch the 2 objects from the current window that are passed to all GDK graphics functions.

`: dirty \ --`
Mark the current window as dirty, which signals the GUI's internal timer to make changes to that window visible. Note that the flag is reset by `redraw`.

26.11.3 Application words

`: COLOR:        \ -- ; --`
Builds a new GTK color. When the color is executed, the foreground color is set. Use in the form:

```
COLOR: red      0xffff0000 , 0xffff w, 0x0000 w, 0x0000 w,
```

The following colours are predefined:

```
red     green   blue    yellow  orange  magenta
cyan    white   black   ltgrey  grey
dkred   dkgreen dkblue  dkyellow brown
violet  dkcyan  dkgrey
```

`: onto          \ window --`
Set the current target window structure for graphics commands.

`: pen           \ -- x y`
Get the current drawing coordinates.

`: at            \ x y --`
Set the current drawing coordinates.

`: filled        \ --`
Makes the next command, such as rectangle or circle, filled instead of stroked.

`: line          { destx desty -- }`
Draw a line from the pen to (destx,desty).

`: lineto        \ destx desty --`
Draw a line from the pen to (destx,desty) and set the pen to (destx,desty).

`: linerel       \ dx dy --`
Draw a line relative to the pen.

`: linerelto     \ dx dy --`

Draw a line relative to the pen and move the pen to the end of the line

`: ellipse       { width height -- }`
Draw an ellipse defined by width, height. The ellipse is positioned such that the pen points to the top left corner of an imaginary rectangle around the ellipse.

`: circle        \ diameter --`
Draw a circle.

The circle is positioned such that the pen points to the top left corner of an imaginary square around the circle.

`: rectangle     { width height -- }`
Draw a rectangle.

`: putpixel      \ --`
Plot a single pixel.

`: cleardevice   \ --`
Clear the current drawing window using the current color.

`: put           { pixbuf -- }`
Draw a Gtk_Pixbuf to the current window at the current pen position.

`: gwin:         \ <name> -- ; -- window`
Declare a named graphics window. The returned window is the address of a `/gwindow` structure.

```
gwin: MyWin    \ -- window
```

`: setupGwin     { w h window -- }`
Initialize a window control structure. This word is used to create a new window. `SetupGwin` cannot be used with windows defined in Glade.

`: initGladeGwin \ z$name builder window --`
Use the Glade widget name (usually a drawing area) in the Glade builder to set up the given `/gWindow` structure.

`: initGwin      \ *widget gwindow --`
Use the widget to set up the given `/gWindow` structure.

`: addEvent      \ cbentry zname event window --`
Add a callback to handle the name and event for a window structure.

`: enable-graphics { window -- }`
Enable the window for graphics operations and set the initial state.

26.12 A text editor in Glade

The original design and C code is by Micah Carrick, whose tutorial is well worth studying. It is at:

```
http://www.micahcarrick.com/gtk-glade-tutorial-part-1.html
```

The Forth code presented here is liberally derived from that presentation and code.

To compile the text editor demo, CD to the directory containing *editor.fth* and then:

```
include TextEdDemo.bld
```

To run the editor from the Forth console:

```
runTextEd
```

The code will run unchanged on VFX Forth for Windows, Mac and Linux.

`struct /TextEd  \ -- len`

Everything we need to know about the editor can be derived from this structure.

`0 value pTextEd \ -- addr`

Holds the address of a structure for the current text editor.

`#1024 constant /NameBuffer      \ -- len`

Largest file name.

`/NameBuffer buffer: zFilenameBuffer     \ -- zaddr`

Buffer to hold current file name.

`#2048 constant /StatusBuffer    \ -- len`

Size of the status buffer

`/StatusBuffer buffer: zStatusBuffer     \ -- zaddr`

Text buffer for status bar.

26.12.1 Tools

`: EdErrBox      \ zmessage --`

Displays an error message box.

26.12.2 Status bar operations

`: sbParams      \ -- sb context`

REturn the status bar parameters

`: setStatus     \ z$1 z$2 --`

set the status buffer, merging the two strings. Then update the status bar.

`: clearStatus   \ --`

Clear the status bar.

26.12.3 TextViews and buffers

`: modified?     \ -- flag`

Return true if the current text has been modified and not saved.

`: modified      \ --`

Mark the current text buffer as modified.

`: unmodified    \ --`

Mark the current text buffer as unmodified.

`: inactive      \ --`

Set the current text window as inactive (unresponsive).

`: active        \ --`

Set the current text window as active (responsive).

`: getCurrText   \ -- text`

Get a copy of the current text. When you are finished with it, you must release it with `g_free ( text -- )`.

26.12.4 Loading and saving text

`: CurrFilename  \ -- z$`
REturn the current text file name.

`: MustSave?     \ -- flag`
Return true if the buffer has been modified and the user wants to save it.

`: getSaveFileName       \ --`
Set the current text file name for saving.

`: getOpenFileName       \ --`
Set the current text file name for loading.

`: sbSaving...   \ --`
Show status bar as saving.

`: sbLoading...  \ --`
Show status bar as loading.

`: fileStatus    \ --`
Show filename on status bar.

`: loadCurrFile  \ -- *text 0 | -1`
Load the contents of the current file. On success, return a pointer to the text and zero. On failure, just return -1. When finished with, the text pointer must be freed with `g_free`.

`: loadCurrText  \ --`
Load the text of the current window from the current file. No action is taken if the filename is null.

`: writeCurrText \ --`
Save the text of the current window to the current file. No action is taken if the filename is null.

`: saveCurrText  \ --`
As `writeCurrText` but asks for a file name if one has not been set.

`: saveAsCurrText        \ --`
As `writeCurrText` but always asks for a file name.

`: checkedSave   \ --`
If text has been modified and the user wants saving, write the text to a file.

`: openCurrText  \ --`
Open a new file.

`: newCurrText   \ --`
Start with an empty buffer.

26.12.5 Clipboard

`: CurrSelection \ -- clipboard`
Get the clipboard item for the current selection.

`: doCut         \ --`
Do the cut operation.

`: doCopy        \ --`
Do the copy operation.

`: doPaste       \ --`

Do the paste operation.

`: doDelete      \ --`
Do the delete operation.

26.12.6 Callbacks

File Menu

`2 0 CCBproc: on_new_MainFileMenu_activate      \ *widget *editor --`
Callback for the "New" button.

`2 0 CCBproc: on_Open_MainFileMenu_activate      \ *widget *editor --`
Callback for the "Open" button.

`2 0 CCBproc: on_save_MainFileMenu_activate      \ *widget *editor --`
Callback for the "Save" button.

`2 0 CCBproc: on_SaveAs_MainFileMenu_activate    \ *widget *editor --`
Callback for the "Save As" button.

`2 0 CCBproc: on_MainWindow_destroy              \ *wiget *editor --`
Callback for final destroy of main window.

`3 1 CCBProc: on_MainWindow_delete_event         \ *widget *event *editor -- 0/1`
When the window is requested to be closed, we need to check if they have unsaved work. We use this callback to prompt the user to save their work before they exit the application. From the "delete-event" signal, we can choose to effectively cancel the close based on the value we return.

`2 0 CCBproc: on_quit_MainFileMenu_activate      \ *wiget *editor --`
Callback for the "Quit" button.

Edit menu

`2 0 CCBproc: on_Cut_MainEditMenu_activate`
Callback for the "Cut" button.

`2 0 CCBproc: on_Copy_MainEditMenu_activate`
Callback for the "Copy" button.

`2 0 CCBproc: on_Paste_MainEditMenu_activate`
Callback for the "Paste" button.

`2 0 CCBproc: on_Delete_MainEditMenu_activate`
Callback for the "Delete" button.

Help menu

`2 0 CCBproc: on_About_MainHelpMenu_activate     \ *widget *user -- ; -- entry`
Callback to run the About dialog.

`2 0 CCBproc: on_aboutdialog1_close      \ *widget *user -- ; -- entry`
Callback when about box is closed.

26.12.7 Initialisation and termination

`: loadTeGUI      \ --`
Load the text editor's GUI. After the file has been loaded, the widgets we need to access are extracted and their object pointers saved in a `/TextEd` structure. The design file object is then released.

```
: termTextEd    \ --
```

free up the application data and perform termination actions.

```
: RunTextEd     \ --
```

Run the text editor.

27 DocGen Documentation Generator

"*Documentation is like sex: when it is good, it is very, very good; and when it is bad, it is better than nothing.*" - *Dick Brandon*

27.1 What DocGen does

DocGen is a simple form of literate programming that enables you to generate software manuals and Forth glossaries directly from the source code of the software. The documentation is produced from formal comments within the source code. The manual for VFX Forth is itself produced this way.

Unlike some other forms of literate programming, all editing is performed directly in the source code itself with the same editor you use for editing the source code. This makes it easy to ensure that your documentation is accurate and up to date.

In order to provide for both on-screen and printed documentation, DocGen can generate output in several forms, referred to as personalities. By default, DocGen is supplied with three personalities. The first, and default, personality generates HTML output. The second is for generating printed documentation, including PDF files, and generates TeX output for use with the Texinfo package. The third personality generates plain Tex output for LaTeX2e. For Windows we use *Miktek* from

```
http://www.miktex.org
```

For Linux use the TeX package supplied with your Linux distribution.

There are a number of packages available for Mac OS X. TexLive is what we use as of 2014.

If you wish to generate output in other forms, instructions for writing additional personalities are provided later in this chapter.

DocGen supports the use of formatting commands within the DocGen formal comments.

The DocGen HTML personality can generate separate output files for a table of contents and an index of documented words. The Texinfo personality can also index documented words using the facilities of Texinfo.

If you are upgrading from a previous version of VFX Forth, see the "Change notes" section at the end of this chapter, and the file *Docs\Release.VFX.txt*.

27.2 Using DocGen

In normal use DocGen is enbled by `+DOCGEN` and disabled by `-DOCGEN` which causes DocGen comments to be generated when source code is compiled.

For use with embedded systems code that cannot be compiled by VFX Forth replace the use of `INCLUDE` by `DOCONLY` or replace `INCLUDED` by `PARSED`. These can also be used to generate manuals separately from the compilation process, which is useful when the ordering of the manual is not the same as the compilation order. Note that the '*>' and '*!' tags described in the section "Marking up your text" can provide fine grain control of ordering, especially for TeX and PDF manuals.

An example of using DocGen for a software project is the manual for the ForthEd2 text editor in *Examples\ForthEd2*. *Examples\ForthEd2\Manual\DocFiles* contains everything needed to generate an indexed manual in both HTML and PDF formats. Feel free to use these as templates for your projects.

DocGen is controlled by the following words.

`defer DocGen-Spacing    \ c-addr u -- c-addr' u'`
If your house rules require DocGen comments to start other than in the first column, assign an action to this `DEFER`red word. The action should remove characters before the start of the comment, returning the modified string as c-addr' u'.

`0 value DocGen?          \ -- flag ; true if DOCGEN enabled`
Returns true if DocGen is enabled, otherwise false.

`: +DOCGEN        \ -- ; enable DOCGEN`
Enables DocGen generation of documentation. After using `+DOCGEN` an output file must be selected using either the '*!' (create) or '*>' (append) tags.

`: -DOCGEN        \ -- ; disable DOCGEN`
Disables DocGen generation of documentation. Note that the active output file is closed, and a new output file must be specified after `+DOCGEN` is used again.

`: PARSED         \ c-addr u --`
Similar to `INCLUDED` but performs no actual compilation. Macros are expanded, and file extensions are resolved using `RESOLVEINCLUDEFILENAME`. `PARSED` allows formal comments to be processed from source code which you do not want compiled, for example embedded systems code for MPE cross compilers. `+DOCGEN` must be used before `PARSED` to enable the DocGen system, and `-DOCGEN` should be used after the last file has been processed. See also `DOCONLY` below.

`: DocOnly        \ "<text>" -- ; DOCONLY <filename>`
Similar to `INCLUDE` but performs no actual compilation. This allows formal comments to be parsed from a source which you do not want compiled, for example embedded systems code for the MPE cross compilers. Use in the form `DOCONLY <filename>`. `+DOCGEN` must be used before `DOCONLY` to enable the DocGen system, and `-DOCGEN` should be used after the last file has been processed.

`: +InternalDocs \ --`
Permits generation of documentation for internal use. DocGen behaves as normal, but also accepts tags of the form +X as well as the normal *X form. This can be used to generate an additional level of documentation above that prepared for normal use.

`: -InternalDocs \ --`
Turns off generation of internal use documentation. Only tags of the form *X will be accepted.

`: +TOC           \ --`
Turn on Table of Contents generation. The table of contents file is called "contents.ext" where ext is defined for the personality being used. The table of contents file is closed by `-DOCGEN`. `+TOC` only affects HTML generation. Tex and Texinfo can already handle table of contents generation.
The table of contents file has four levels, corresponding to the T, S, N and H tags.

`: +Index \ --`
Turn on Index generation. The HTML index file is called "indices.html". The index file is closed by `-DOCGEN`. For Texinfo and Latex2e the index entries are placed in the output and the index is generated by Texinfo and LaTex. At present `+Index` only affects HTML and Texinfo generation.
The HTML index file can later be sorted using the command line **SORT** utility which is still present in all versions of Windows. Under Windows XP, the command line should be:

```
sort /+51 indices.html /O sorted.html
```

DocGen produces *indices.html* with word names starting in column 51. The output of the **SORT** utility is the file *sorted.html*.

The command line:

```
sort /?
```

will display the command syntax.

N.B. The index file contains no header or footer text, e.g. <HTML>, because the sorting process will move these in the file. After sorting, the file should be edited to add suitable headers and footers. For HTML output, examples follow.

```
<HTML><BODY bgcolor="#C2C1B4">
<BR><BR><HR><BR><H1><font color="#ff0000">
Index
</font></H1><BR><HR><BR><BR>
  ...
</HTML>
```

Texinfo and Latex indices are sorted by the *makeindex* program. You do not need to call this yourself as it can be handled by the *texify* program. See the examples at the end of the chapter for more details.

27.3 Marking up your text

DocGen output is derived from formal comments within the source code. The output format defaults to standard HTML 2.0. The layout of the final document is controlled by DocGen tags at the start of the comment and by formatting macros within the body of the comments.

27.3.1 Comment tags

The DocGen comment takes one of the following two forms:

```
( *x blah blah )
```

where the open bracket must be in the first column, the closing bracket must be the last character, and X is the operation code or tag and "blah blah" is the control text.

```
\ *x blah blah
```

where the backslash character '\' must be in the first column, and X is the operation code or tag and "blah blah" is the control text.

In either case there must be one space between the comment marker and the asterisk. If you are using `( *x ...)` comments there must be at least one space after the tag operation code, even if no additional text follows it. The '*X' combination is referred to as a DocGen tag.

Tag codes and their actions

`*` The following text is a continuation for the current style. This can only be used after a G, E, D, B, Q or P tag.

`!` Create and select a new output file. The control text is the filename without extension.

`>` Select and append to an existing output file. The control text is the filename without extension.

`T` Following text is a section title.

`S` Following text is a section sub-title.

`N` Following text is a section sub-sub-title.

`H` A simple heading.

`D` Following text is a definition. The first space delimited token is the term, the remaining text the description.

`P` Begins a new paragraph.

`E` Begins a paragraph which is a code example.

`(` Starts a bullet list. The first character of the text defines how the lists are marked. No text produces bullets, **1** produces a numbered list and **A** or **a** produce lettered lists. Lists may be nested. **N.B.** Previous markups that did not use the ***(** and ***)** tags still function as before, but the new layout is generally visually better. Note that the LaTeX personality only handles bulleted and numbered lists. See the change notes section of this chapter for more details.

`B` Following text is a bulleted entry.

`)` Ends a bullet list.

`G` Following text is a glossary entry. The preceeding line is output in a fixed font code format.

`R` Following text is output directly when using TeX output.

`W` Following text is output directly when using HTML output.

`C` A fixed font line, e.g. a code example

`Q` A quotation which may extend over several lines.

`[` The following code will also be copied into the documentation. DocGen lines are handled as usual. You **must** use the *] tag to stop copying - it is not halted by other tags. Do not use any other tags before copying is stopped.

`]` Stops copying code into the documentation.

`L` Inserts a line break, full width horizontal line and a line break. Any text after the tag is ignored.

Please examine MPE supplied sources for a view of how DocGen is used to build library and API documentation.

Tags reserved for DocGen/SC

The following tags are reserved by MPE for use with DocGen/SC for safety critical applications.

```
*O *A *V *U *M *X *Y *Z
```

If you extend DocGen yourself, avoiding these tags will help to ensure compatibility with future versions of DocGen. DocGen/SC is described later in this chapter.

27.3.2 Formatting macros

From VFX Forth v3.70 onwards DocGen supports formatting commands that can be included inside DocGen comments. Commands are of the form

```
... *\command{text} ...
```

The *text* is processed and output. The following commands are supported in all three default personalities.

`bold` Displays text in **bold**.

`b` as bold

`fixed` Displays text in a `fixed (typewriter)` font.

`f` as fixed

`italic` Displays text in *italic*.

`i` as italic

`forth` Displays text in **`bold and fixed`**.

`fo` as forth

`br` generates a line break. The text is ignored.

`starbslash`
Displays `*\`. The text is ignored.

These commands are used to change the formatting of small pieces of text and make it easier to identify what is being referred to. For example, a reference to a Forth word can be produced by:

```
\ ** the word *\forth{DUP} is often used
```

which produces the result:

```
\ ** the word DUP is often used
```

27.3.3 Table macros

Tables are described using DocGen macros, which are **not** present in the LaTex personality.

`table{list}`
Starts a table. The *list* contains an optional caption delimited by '"' characters, followed by space separated numbers. These are the percentage widths of each

column in the table, and so the list also defines the number of columns in the table. There may be up to 16 columns.

`endtable{}`
Marks the end of the table.

`hrow{}` Marks the start of a header row. Header rows are formatted differently from normal rows.

`row{}` Marks the start of a normal row.

`col{}` Marks the start of a new column in a row - you do need this for the first column.

27.3.4 Image macros

An image (graphics file) is incorporated using the `image` macro, which is **not** present in the LaTex personality. It is used in the form:

```
*\image{"caption" "file" "ext" hmm w%}
```

where *caption* is the text of the image caption; *file* is the basic file name with no extension; *ext* is the file extension; *hmm* is the image height in millimetres for Texinfo and LaTex; and *w%* is the image width in percent for HTML. All five parameters must be present.

Not all personalities will use all the parameters. For example, when using *pdftex* the file extension may be unused, and *pdf* images are always used. When using HTML output the *hmm* parameter is ignored, and the image size is taken from the percentage width.

When preparing PDF images for use with *pdftex*, note that most tools generate PDF pages. Consequently, when preparing an image for export as a PDF image, first select portrait or landscape page format as appropriate, then scale the image to fit the page, and then save it as a PDF file. OpenOffice Draw is a suitable tool for converting most graphics files into PDF format.

27.4 Defining a new personality

You can extend the DocGen system yourself. Many utility words are available in the module `DOCGEN`. Because of the number of words you will have to write, we recommend that new personalities be defined in a `VOCABULARY` or a `MODULE`. Use the DocGen source code in *Sources\Lib\Docgen.fth* as a model.

WARNING: From build 1720 onwards DocGen specific utility words defined in the `DOCGEN` module are no longer `EXPORT`ed. You must now surround the code that uses them with the following fragment:

```
expose-module docgen
  ...
previous
```

27.4.1 Personality description notation

Defining a new personality consists of naming it, defining the file extension that will be added to the output file name, and then defining the tags that the personality will react to. It is recommended that you provide actions for all the tags provided as default, so that DocGen will still be able to generate documentation using any of the predefined personalities. The example below is for the default HTML personality.

```
[DocGen Docgen_html
  DG_FileExt: html

\ the continuation tag    entry   exit    continuation
  DG_Tag: *       0       illegal illegal illegal

\ tags whose text must fit on one line
  DG_Tag: !       -1      >newf   illegal illegal
  DG_Tag: >       -1      >tofl   illegal illegal
  DG_Tag: T       -1      >titl   illegal illegal
  DG_Tag: S       -1      >sect   illegal illegal
  DG_Tag: N       -1      >sstl   illegal illegal
  DG_Tag: H       -1      >head   illegal illegal
  DG_Tag: R       -1      2drop   illegal illegal
  DG_Tag: W       -1      >rweb   illegal illegal
  DG_Tag: C       -1      >code   illegal illegal

\ tags that can have continuation lines
  DG_Tag: D       1       >defn   defn>   defn
  DG_Tag: P       1       >para   para>   para
  DG_Tag: E       1       >exam   exam>   exam
  DG_Tag: B       1       >bult   bult>   bult
  DG_Tag: G       1       >glos   glos>   glos
  DG_Tag: Q       1       >quot   quot>   quot

DG_Type: HTMLtype

[DG_Macros
: bold              \ caddr u --
  ." <B>"  prepro HTMLtype  ." </B>"
;

: fixed             \ caddr u --
  ." <TT>"  prepro HTMLtype  ." </TT>"
;

: italic            \ caddr u --
  ." <I>"  prepro HTMLtype  ." </I>"
;

: forth             \ caddr u --
  ." <B><TT>"  prepro HTMLtype  ." </TT></B>"
;

: starbslash        \ caddr u --
  2drop  S" *\" HTMLtype
;
DG_Macros]

DocGen]
```

[DOCGEN <name> defines a new personality. When <name> is executed, it becomes the current personality.

DG_FileExt: <ext> specifies the extension that will be added to the output file names for this personality.

DG_Tag: specifies the character, control code, and actions of a tag. The given character is used as the second character of the '*x' pair. The second entry is the control code and specifies the state conditions required, see later. The next three entries are the names of the Forth words executed on entry to the tag, on exit from that tag when another tag is selected, and how that tag line should be handled. See later for how the three action words should be defined.

DG_Type: <name> specifies that <name> is the word used for TYPEing output.

The pair [DG_Macros and DG_Macros] delimit command words defined in the personality's private wordlist. These provide the actions of the command macros.

DOCGEN] marks the end of the personality description.

27.4.2 Using control codes

DocGen operates using three states - entry, exit and continuation. When a new tag is encountered, the previous tag's exit code is run if the control code is non-zero. The exit code can for example add a </P> paragraph end marker to HTML code.

If the tag's control code is 1, the new tag becomes the currently active tag so that the '**' tag knows what to do, and the entry code for the new tag is run. This is the normal condition for a tag that can have continuation lines.

If the tag's control code is -1, the entry code is run, and the currently active tag state is set to none. A condition code of -1 is used for tags whose following text must fit on a single line, for example section titles.

If the tag's control code is 0, the continuation action of the previous tag is run.

27.4.3 Writing the action words

The stack comments of the three action words are as follows. Text output of words such as EMIT and TYPE has already been set to go to the last defined output file.

```
: entry      \ caddr u -- ; consumes the text for the line
: continue   \ caddr u -- ; consumes the text for the line
: exit       \ -- ; close the tag
```

The utility word PREPRO is part of the primary definition and converts any special characters in the DocGen comments. PREPRO has the stack effect:

```
caddr u -- caddr' u'
```

where caddr u defines the input text and caddr' u' defines the output text. The MPE handlers use the following special characters apart from those special to the output format itself.

`0x09` tab character, convert to spaces, tab width is set to 8 characters

`0x1E` convert to open bracket character

`0x1F` convert to close bracket character

The following code is for the glossary `*G` tag in the HTML output. The word `LASTDEFINITIONLINE` returns the text for the previous line.

```
: >glos         \ caddr u --
  cr ." <PRE><CODE><B>" LastDefinitionLine prepro type
  ." </B></CODE></PRE><P ALIGN=JUSTIFY>"
  cr prepro type
;

: glos          \ caddr u --
  cr prepro type
;

: glos>         \ --
  cr ." </P>" cr
;
```

The utility words `NewDocFile` and `SwitchDocFile` are provided to aid construction of the *! and *> tags which create a new file or append to an existing file. Both have the stack comment:

```
caddr u --
```

which is the file name without extension. NewDocFile starts a new file, and you can add any required file entry code as required. SwitchDocFile switches to the end of the named file.

The word `PREPRO` is available for processing text for tags and formatting macros. It has the stack comment

```
caddr len -- caddr' len'
```

The input text is processed and passed to an output buffer. Special DocGen charactes (TAB, 0x1E, 0x1F) are converted and format commands are processed.

27.4.4 Formatting commands

DocGen supports formatting commands that can be included inside DocGen comments. Commands are of the form

```
... *\command{text} ...
```

Each command is handled by a Forth word defined in a private `wordlist` for the personality. The **command** word is passed the string text, and has the stack comment

```
caddr len --
```

Formatting commands use ." and friends for output. The text defined by caddr/len must be processed by PREPRO before output. The string returned by PREPRO must be output by a version of TYPE specific to the personality.

```
caddr len --
```

The HTML version, called HTMLtype outputs the text converting special characters to the form required by the output format, e.g it must handle '<' and '>' for HTML and '\' for Tex.

27.4.5 Personality words glossary

: [DocGen \ "<spaces>name" -- ; -- ; start new DOCGEN personality
Defines the start of a new personality for DocGen. See the example above for details of the use of [DOCGEN and DOCGEN].

: DocGen] \ -- ; end personality
Marks the end of a new personality for DocGen. See the example above for details of the use of [DOCGEN and DOCGEN].

: DG_FileExt: \ "<spaces>text" -- ; define document file extension
Defines the file extension for the DocGen personality being defined. See the example above for details of the use of DG_FILEEXT:.

: DG_Tag: \ "<char>" "<code>" "<enter>" "<exit>" "<continue>" --
Defines how a tag character is handled by DocGen. See the example above for details of the use of DG_TAG:

: DG_Type: \ "<word>" --
Specifies the word which performs the action of TYPE (addr len –) for this personality. Special characters are translated, e.g. in HTML the '<' character is issued as "<". Use in the form:

```
DG_Type: <name>
```

: [DG_Macros \ -- oldcurrent
Marks the start of defining formatting macros.

: DG_Macros] \ oldcurrent --
Marks the end of defining formatting macros.

: .DG_Macros \ --
Show the available formatting macros in the current personality.

: .DG_Tags \ --
Show the available tags in the current personality.

27.5 HTML output

This personality generates HTML output.

[DocGen Docgen_html \ -- ; -- ; select HTML for DOCGEN
Makes the HTML personality the current personality for DocGen. HTML will remain the current personality until another personality is selected.

: HTMLback \ caddr len --
Sets the HTML background colour for the next output file. The string is the HTML colour reference (limited to 31 characters), e.g.

```
s" #00C1B4" HTMLback
```

27.6 TeX output with texinfo.tex

This personality generates TeXinfo output, using the file texinfo.tex supplied with many TeX distributions. From TeX you can get to many other formats including PDF.

TeXinfo is under-documented, especially in terms of examples, and has quirks. Despite this, Texinfo can generate good quality PDF files with bookmarks and thumbnails, a table of contents and an index. The MikTex CD package (see below) includes the Texinfo manual file *doc\texinfo\texinfo.dvi* which can be viewed using *bin\yap.exe*. Texinfo is a real "techie" extension to LaTex, but the more we use it, the more we appreciate it. It is worth the admittedly steep learning curve. Texinfo code can be found in the examples at the end of this chapter.

Glossary entries (the ones with `*G` tags are indexed. Additional indexing macros will be added in a future release.

For Windows, the MikTeX package should be suitable for use with the output of DocGen and the file *texinfo.tex* is required. Version 2.4 of MikTeX is supplied on the VFX Forth CD. In addition, a converter from the TeX DVI output format to PDF will be required if PDF manuals are to be generated and if you do not have *pdftex* or *pdflatex*. MikTeX includes *pdflatex* which is a version of LaTeX that produces PDF files directly. Most distributions include *pdftex* which performs the same operation and may be more suitable for TeXinfo than *pdflatex*.

The MikTeX home page is at www.miktex.org. Note that the v2.4 small distribution is over 24Mb. For other distributions, which are much larger, a CD is available from the home page for a small charge. We encourage you to get the CD of the latest release if you intend to use Texinfo.

If you are using the v2.4 distribution supplied by MPE, install the MikTex package by running the installer. Make sure that the *MIKTEX\BIN* directory is in your search path, and generate a master document file using *MANUAL.TEX* as a template. Run DocGen with the `DOCGEN_TEXINFO` personality to produce your output files, and make sure that you have included lines of the form `@`*include file.tex* for each file in the manual. Then run *LATEX.EXE MANUAL* to produce a DVI file or *PDFLATEX.EXE MANUAL* to produce a PDF file.

`: lpc             \ char -- char'`
Convert char to lower case if it was alphabetic.

`[DocGen DocGen_TexInfo  \ -- ; -- ; select TeX personality`
This word makes the TexInfo personality the current personality for DocGen. TexInfo will remain the current personality until another personality is selected.

27.7 LaTeX2e output

This personality generates LaTeX2e output. From LaTeX2e you can go to many other formats including PDF. See the previous TeX section for details of the required associated tools and their installation.

27.7.1 Installation

Install the MikTex package by running the installer. Make sure that the MIKTEX\BIN directory is your search path, and generate a master document file using MANUAL.TEX as a template. Run DocGen with the DOCGEN_LATEX personality to produce your output files, and make sure that you have included lines of the form "\include{file}" for each file in the manual. Then run "LATEX.EXE MANUAL" to produce a DVI file, or "PDFLATEX.EXE MANUAL" or "PDFTEX MANUAL" to produce a PDF file.

27.7.2 Basic usage

The output of DocGen is a set of LaTex2e files as defined by the tags above. The manual is then generated by processing these files with PDFLATEX.EXE. An example of MANUAL.TEX is given below. Each file that you want to process should have a line of the form "\include{filename}" where the .TEX extension will be assumed.

```
\documentclass[a4paper, 10pt]{book}
\parindent 0pt
\parskip 1ex plus .5ex
\begin{document}
\include{docgen}
\end{document}
```

The output of running PDFLATEX MANUAL will be file called MANUAL.PDF ready for distribution. There will also be a large number of MANUAL.* temporary files, all of which can be deleted as required. The only one of potential interest is MANUAL.LOG, which contains the error report. In most cases, there will be many reports of the form "Overfull \hbox" which can be ignored.

You can configure the appearance of the manual by editing MANUAL.TEX as you require. LaTeX is a very powerful document processing system, and you can modify nearly everything, as well as add indices and so on. Several books about LaTeX and TeX are available from Amazon, and the best source of information about the system is at www.tex.ac.uk which will point you at implementations for many machines as well as several tutorial packages.

27.7.3 Adding a title page

Adding a title page requires a few more lines, and an example is given below.

```
\NeedsTeXFormat{LaTeX2e}
\documentclass[a4paper, 10pt]{book}
\parindent 0pt
\parskip 1ex plus .5ex
\begin{document}
\author{MicroProcessor Engineering Ltd}
\title{DOCGEN/SC}
\maketitle
\include{docgensc}
\end{document}
```

27.7.4 Adding a Table of Contents

Adding a table of contents requires only a few more lines, and an example is given below.

```
\NeedsTeXFormat{LaTeX2e}
\documentclass[a4paper, 10pt]{book}
\parindent 0pt
\parskip 1ex plus .5ex
\begin{document}
\author{MicroProcessor Engineering Ltd}
\title{DOCGEN/SC}
\maketitle
\pagenumbering{roman}
\tableofcontents
\newpage
\pagenumbering{arabic}
\include{docgensc}
\end{document}
```

Note that in order to generate a table of contents, LATEX needs to be run twice. On the first run, the table of contents will be inaccurate as the table of contents file will be the one from the previous run which is unlikely to match the new first run. After the second run on the same files, the table of contents will be accurate.

`[DocGen DocGen_LaTex    \ -- ; -- ; select LaTeX personality`

This word makes the LaTex personality the current personality for DocGen. LaTex will remain the current personality until another personality is selected. See the previous section about TeX for details of the system requirements.

27.8 DocGen kernel hooks

The VFX Forth Kernel contains `DEFER`ed hooks at strategic points within the compiler/interpreter. These hooks are used by DocGen to install and uninstall itself. The DocGen hooks are:-

`defer DOCGEN_PREREFILL  \ --`

Called within `REFILL` before another line of text is read.

`defer DOCGEN_REFILL     \ --`

Called within `REFILL` after the next text line has been read. at this point you may process the INPUT buffer (from `SOURCE`) but you must **not** under any circumstances change it.

`defer -DOCGEN_HOOK      \ --`

Disables DocGen without closing the active output file. This is useful for conditional generation of documentation, particularly if several versions of the software exist. Note that files containing such phrases must be `INCLUDE`d, as `DocOnly` turns off the Forth interpreter.

```
[undefined] <someword> [if]  -docgen_hook  [then]
\ ** This documentation is needed if <someword>
\ ** exists ...
+docgen_hook
```

`defer +DOCGEN_HOOK      \ --`

Enables DocGen output again. See the previous example

27.9 Organising Manual generation

You can make manual generation very much easier by creating some auxiliary files which manage the process.

- DocGen control file, `INCLUDE`d by VFX Forth to create the HTML and/or Tex files. This file often also `INCLUDE`s the file below. To distinguish it, we usually give this file a ".DGS" extension.
- List of source files to process. If you are creating both HTML and PDF documentation, the process is simplified by using the same list for both generation phases. This file is usually called something like *jobfiles.fth*.
- A batch file to run the whole process.

Examples are given of all of these files, somewhat attenuated to remove obvious repetition. The examples are taken from the documentation for the MPE ARM USB Stamp on-board software. Both HTML and PDF files are produced.

These examples contain several "MPEisms", in particular the use of *###* files. After DocGen has finished, we will find files *###.html* and *###.tex*. If these files have anything in them, some documentation has been missed. When a source file ends, the following section adds further DocGen output to the junk file:

```
\ ======
\ *> ###
\ ======
```

At the start of each source file, you must declare where output goes using one of the '*!' or '*>' tags. Thus the *###* junk files collect any documentation that has been written but not associated with an output file. Looking at the junk files tells you if this has happened.

27.9.1 Sample DocGen Control file

There are two sections to this file, one for the HTML documentation and one for the Texinfo code that then creates the PDF documentation.

```
\ USBSTAMP.DGS - ARM USB Stamp DOCGEN control file

\ turn on DOCGEN and select personality
+docgen  docgen_html

cr ." Starting HTML manual generation" cr
```

The following section produces the main HTML file with a left-hand chapter selection menu.

```
\ *! index
\ *W <HEAD><TITLE>MPE ARM Forth Stamp Code Manual</TITLE></HEAD>
\ *W <FRAMESET COLS="200, 100%">
\ *W   <BODY bgcolor="#C2C1B4">
\ *W   <FRAME SRC="menu.html"       name="menu">
\ *W   <FRAME SRC="stamptitle.html" name="main">
\ *W   </BODY>
\ *W </FRAMESET>
\ *W </HTML>
\ *> ###
```

Then we can create the chapter selection menu, which references the HTML files produced by DocGen.

```
\ *! menu
\ *W <A target="main" HREF="stamptitle.html" >Home</A><BR>
\ *W <A target="main" HREF="intro.html"      >Introduction</A><BR>
\ *W <A target="main" HREF="codearm.html"    >Low level Kernel</A><BR>
  ...
\ *W <A target="main" HREF="romforth.html"   >ROM FORTH extensions</A><BR>
\ *W <A target="main" HREF="xmodemtxrx.html" >XModem File Transfers</A><BR>
\ *> ###
```

We can create a title page.

```
\ *! stamptitle
\ *W <TABLE border=0 cellPadding=0 cellSpacing=0 width="100%">
\ *W   <TBODY>
\ *W   <TR><TD>
\ *W     <IMG src="mpelogo.gif">
\ *W   </TD></TR>
\ *W   </TBODY>
\ *W </TABLE>

\ *W <CENTER>
\ *W <H1>MPE ARM Stamp Software Reference Manual</H1>
\ *W <P>7 October 2004</P>
\ *W <P><I>Documentation derived from the source code by DOCGEN
\ *W       with VFX Forth for Windows
\ *W </I></P>
\ *W <BR><BR><BR>
\ *W <B>(C)opyright 2004 MicroProcessor Engineering Limited.</B>
\ *W </CENTER>
\ *> ###
```

Now we can generate all the other files from the second file. Because DocGen automatically adds a colour to the start of the *INDEX.HTML* file, we include a comment to remove it because it will be in the wrong place when frames are used.

```
include stampfiles

cr ." HTML Manual generation done" cr

cr
cr ." **********************************"
cr ." Remove the BODY tag in the first line of INDEX.HTML"
cr ." **********************************"
cr

-docgen
```

The procedure is essentially the same for the PDF documentation, except that the table of contents is produced by Texinfo.

```
\ turn on DOCGEN and select personality
+docgen  docgen_texinfo

cr ." Starting Texinfo manual generation" cr


\ ===============
\ Tex manual file
\ ===============
\ *! manual
\ *R \input texinfo
\ *R @setfilename           usbstamp.info
\ *R @setcontentsaftertitlepage
\ *R @afourpaper
\ *R @settitle               MPE USB Stamp
\ *R @setchapternewpage      odd
\ *R @paragraphindent        0
\ *R @exampleindent          0
\ *R @finalout

\ *R @include titlepg.tex
\ *R @include intro.tex
\ *R @include codearm.tex
  ...
\ *R @include romforth.tex
\ *R @include xmodemtxrx.tex

\ *R @bye
\ *> ###

include stampfiles

cr ." Texinfo Manual generation done" cr

-docgen
```

If you are generating an index, add the following three lines before the line containing "@bye"

```
\ *R @unnumbered{Index}
\ *R @*
\ *R @printindex fn
```

27.9.2 Example file list

The first section defines a set of text macros to reduce typing and ease changes when the project directories (folders) are moved.

```
\ STAMPFILES.FTH - DOCGEN include files

c" c:\buildkit.dev\software\rom\arm"
  setmacro CpuDir
c" c:\buildkit.dev\software\rom\common"
  setmacro CommonDir
c" c:\buildkit.dev\software\rom\examples"
  setmacro ExampleDir
c" c:\buildkit.dev\software\rom\arm\hardware\LPC210x"
  setmacro StampDir
```

Now comes the list of files. This file was started before the word `DocOnly` was available. The .FTH extension is not required as the smart file include system will try a range of extensions. MPE habit is to give files that only contain DocGen comments a .MAN extension.

```
s" intro.man" parsed
s" %CpuDir%\codearm.fth" parsed
s" %CommonDir%\kernel62.fth" parsed
  ...
s" %CommonDir%\voctools.fth" parsed
s" %CommonDir%\xmodemtxrx.fth" parsed
DocOnly romforth.man
DocOnly examples.man
DocOnly titlepg.man
```

27.9.3 Example batch file

The batch file controls the whole operation and removes the collection of temporary files produced by PDFTEX or TEXIFY. The "start /w" command is used to make the batch file wait until a GUI program has finished.

Note that if an index is being generated by Texinfo, the line

```
pdftex manual
```

must be replaced by

```
texify -p manual.tex
```

where the extension must be supplied.

```
@echo off
rem B.BAT controls the operation
echo *****************************
echo Date changed in USBSTAMP.DGS?
echo *****************************
del *.html
del *.tex
start /w c:\products\pfwvfx\bin\pfwvfx include usbstamp.dgs

echo Error log will be in manual.log
del manual.log
echo starting pass 1 ..
pdftex manual
echo .. pass1 complete, starting pass 2 ..
pdftex manual
echo .. pass 2 completed

pause
move manual.pdf USBStampCode.pdf
echo Manual is in file USBStampCode.pdf

echo Deleting temporary files
move manual.log temp.log
del manual.*
move temp.log manual.log


echo ***********************************
echo Modify the first line of INDEX.HTML
echo ***********************************

echo Press Control-C not to issue manuals
pause
del ###.*
del ..\*.html
del *.tex

move *.html ..
move *.pdf ..
copy *.gif ..

echo Done
pause
```

27.9.4 Example Texinfo title page

This example only applies to the Texinfo personality.

```
\ ==========
\ *! titlepg
\ ==========
\ *R @titlepage
\ *R
\ *R @title                 MPE ARM USB Stamp
\ *R @subtitle              v1.0
\ *R @author                Microprocessor Engineering Limited
\ *R @page
\ *R
\ *R @vskip 0pt plus 1filll
\ *R
\ *R Copyright @copyright{} 2004 Microprocessor Engineering Limited
\ *R
\ *R Published by Microprocessor Engineering
\ *R
\ *R
\ *R
\ *R @page
\ *R
\ *R MPE ARM USB Stamp                      @*
\ *R User manual                            @*
\ *R Manual revision 1.00                   @*
\ *R @today{}                               @*
\ *R                                        @*
\ *R                                        @*
\ *R Software                               @*
\ *R Software version 6.20                  @*
\ *R                                        @*
\ *R                                        @*
\ *R For technical support                  @*
\ *R Please contact your supplier           @*
\ *R                                        @*
\ *R                                        @*
\ *R For further information                @*
\ *R MicroProcessor Engineering Limited     @*
\ *R 133 Hill Lane                          @*
\ *R Southampton SO15 5AF                   @*
\ *R UK                                     @*
\ *R                                        @*
\ *R Tel:       +44 (0)23 8063 1441         @*
\ *R Fax:       +44 (0)23 8033 9691         @*
\ *R e-mail:    mpe@@mpeforth.com     @*
\ *R   tech-support@@mpeforth.com     @*
\ *R web:       www.mpeforth.com      @*
\ *R                                        @*
\ *R @page
\ *R @end titlepage
\ ======
\ *> ###
\ ======
```

27.10 Change notes

From VFX Forth v3.90 build 2020, '\ *x' lines with no following text do not require a trailing space at the end of the line - it caused too many problems for us. The HTML personality no longer generates `<BODY bgcolor="#C2C1B4">` after `<HTML>` if the output file is called **index**. The HTML background colout can be set by using the word `HTMLback ( caddr len -- )` **before** the file is created, e.g.

```
s" #C2C1B4" HTMLback
```

Line breaks can be forced using the **br** macro.

Bulleted lists can now be nested and there is a choice of bullet styles. Lists are **optionally** surrounded by the `*( text` and `*)` tags. The first character of the opening text can be null for bullets or one of `1`, `A` or `a` for enumerated or lettered lists. If the `*( text` and `*)` tags are used, lists may be nested.

As of VFX Forth v3.70, DocGen supports the use of formatting commands within the DocGen formal comments. The Texinfo default formatting has been changed to improve the look of PDF manuals. Because of these changes, personalities for earlier versions of DocGen must be updated. See the section on writing personalities for more details.

As of VFX Forth v3.70 build 1720, the DocGen HTML personality can generate a table of contents and an index of documented words. The Texinfo personality can also index documented words.

27.11 DocGen/SC

DocGen/SC is an extension of DocGen for documenting safety critical systems. DocGen/SC allows test code to be provided after each word and to be extracted to separate test files, so automating the production of regression tests.

DocGen/SC produces documentation that has been accepted by organisations such as the US FDA for medical equipment. The documentation format and test files are also suitable for other authorities and application domains including avionics and transport systems. Contact MPE for more details.

All the tags of DocGen work as they did before. Some new tags have been defined to control the safety critical documentation process.

`O` followed by "initials" "name" "organisation". A list of authors is kept and authors only need to be defined once.

`A` followed by "initials" selects the current author. Once selected, the author's name and organisation will be output for each word definition.

`V` followed by "version_text" sets the version information produced in the header for each definition.)

`U` followed by "10" sets the width of hard tabs, ASCII code 9, to the given integer value. This value defaults to 8, and is used when expanding tab characters in the source code and test code output. In general, we recommend that hard tabs are always set to 8 characters as this is the default value used by many applications.

`M` starts the notes section of the output.

`X` followed by `"filename"` defines the file used to contain the test code. This file is closed after each source file is `PARSED` and each source file should select a test file into which the code between `[TEST` and `TEST]` is to be placed.

`Y` marks the start of the test code section.

`Z` marks the end of the definition and triggers checks.

28 Library files

The *Lib* folder/directory contains tools maintained and and periodically updated by MPE. The contents of *Lib* differ between the Windows, Linux, OS X and DOS versions as some of the tools are operating system specific.

28.1 Building cross references

28.1.1 Introduction

Cross reference information helps you to manage your source code. When *LIB\XREF.FTH* is loaded you can use `XREF <name>` to find out in which other words `<name>` is used. You can also find out which words you defined but did not use. `XREF` is precompiled in the Studio version of VFX Forth but not in the base version.

The compiler generates cross references by building a chain of fields including `LOCATE` format (link:32, xt:32, line#:32) in a separate area of memory. Links and pointers are relative to the start of the `XREF` memory area.

Two chains are maintained. The first produces a chain of where a word is used, so that the user can find out where (say) `DUP` is used. The second produces a chain of which words and literals are called in order. This is the basis of decompilation and debugging.

28.1.2 Initialisation

`XREF` is initialised by the switch `+XREFS` and is terminated by `-XREFS`. You must use `+XREFS` to turn on the production of cross reference information.

By default 1Mb of cross reference memory is allocated from the heap. If you need more than this for a very large application, use the phrase `<n> XREF-KB` to set the size of the cross reference memory, where `<n>` is in kilobytes.

28.1.3 Decompilation and SHOW

Because the VFX code generator optimises so heavily, there is no direct relationship between the binary code and the source code. Consequently `DIS` and `DASM` use disassembly and special cases, but cannot produce a good approximation to the original source code.

The cross reference information includes a decompilation chain. When you use `SHOW <name>` the cross reference information is used to produce a machine decompilation. This includes none of the comments from the original source code, and is machine formatted.

28.1.4 Extending SHOW

The decompilation produced by `SHOW` is mostly default and automatic. However, some words such as string handling take in line data which would not be displayed by `SHOW` without special handling.

`SHOW` can be extended by adding items to the `DCC-SWITCH` chain. The stack effect of the action is: addrx – addr ; where addrx is the offset of the cross reference packet in the cross reference information memory. See the `/REF[X]` structure in *LIB\XREF.FTH* for details of the structure of this data packet. The example below is for a word X" which takes an in-line string like S".

```
[+switch dcc-switch
  ' X"       run:  ." X"  [char] " emit  dup .$inline  ;
switch]
```

Note that unlike previous VFX Forth decompilers, `SHOW` is based on cross reference information which references the source word without knowledge of what it compiles. The only reasons for special cases are control of the decompilation layout and display of associated data to reconstruct source code.

28.1.5 Glossary

`: dump(x)        \ offset len --`

Displays the specified contents of the XREF table. Note that the given address is an **offset** from the start of the XREF table.

`: init-xref      \ --`

Initialise XREF memory and information if not already set up.

`: term-xref      \ --`

Free up XREF memory.

`: save-xref      \ -- ; save XREF memory to file`

Save the cross reference memory to disc. Unless the file name has been changed by `XREF: <filename>` the file will be called *XREF.XRF*.

`: load-xref      \ -- ; reload XREF file from disc`

Load the cross reference memory from disc. Unless the file name has been changed by `XREF: <filename>` the file used will be *XREF.XRF*.

`: xref:          \ "filename" -- ; enable XREFs`

Use in the form `XREF: <filename>` to define the file that `SAVE-XREF` and `LOAD-XREF` will use.

`: xref-kb        \ n --`

Specifies the size of the cross reference memory in kilobytes. By default this is 1024 kb, or 1Mb.

`: +xrefs         \ -- ; enable XREF`

Initialises the cross reference system if it has not already been initialised, and enables production of cross reference information.

`: -xrefs         \ -- ; disable XREF`

Stops production of cross reference information, which can be restarted by `+XREFS`. Cross reference memory is not erased or released. Thus, restarting with `+XREFS` will retain information. To release all previous information use `TERM-XREF` before `+XREFS`.

`: xref-report    \ -- ; display XREF information`

Displays some statistics about cross reference memory usage.

`: WalkXref       \ xt1 xt2 -- ; XREF of XT1 using XT2 to display.`

Used by application tools to walk the XREF chain for XT1. The structure offset for each step in the chain is handled by XT2 (offset –). Because writing XT2 requires use of the internal XREF structure, you must expose the `XREFFER` module: `EXPOSE-MODULE XREFFER` to get access to the words in *Lib\XREF.FTH*.

`: (show)         \ xt -- ; show/decompile words used by this XT`

Given an XT, produces a machine decompilation of the word using the cross reference information. If cross referencing is not enabled, no action is taken.

```
: $show         \ $addr --
```

Given a counted string, it is looked up as a Forth word name and (SHOW) produces a machine decompilation of the word using the cross reference information. If cross referencing is not enabled, no action is taken.

```
: show          \ -- ; SHOW <name>
```

The following name is looked up as a Forth word name and (SHOW) produces a machine decompilation of the word using the cross reference information. If cross referencing is not enabled, no action is taken.

```
: hasXref?      \ xt -- flag ; true if word has XREF info
```

produces TRUE if xt has XREF information otherwise FALSE is returned.

```
: hasXDecomp?   \ xt -- flag ; true if word has XREF decompilation info
```

produces TRUE if xt has XREF decompilation information otherwise FALSE is returned.

```
: WalkDecomp    \ xt1 xt2 -- ; DECOMP of XT1 using XT2 to display.
```

Used by application tools to walk the decompilation chain for XT1. The structure offset for each step in the chain is handled by XT2 (offset –). Because writing XT2 requires use of the internal XREF structure, you must expose the XREFFER module: EXPOSE-MODULE XREFFER to get access to the words in *Lib\XREF.FTH*.

```
: FindXrefInfo  \ pc xt -- info | 0 ; finds xref packet corresponding to PC
```

Given the current PC and the XT of the word the PC is in, FindXrefInfo returns a pointer to an XREF packet if the PC is at an exact compilation boundary, otherwise it returns zero.

```
: FindXrefNearest       \ pc xt -- info|0
```

Given the current PC and the XT of the word the PC is in, FindXrefNearest returns a pointer to the Xref packet for the address at or less than the PC. If no Xref information is available for the word, zero is returned.

```
: GetXrefPos    \ info -- startpos len line addr
```

Given a pointer to an XREF packet, GetXrefPos returns the position, name length, line number of the source text in the source file, and the value of HERE at the time of compilation.

```
: NextXref      \ info1 -- info2
```

Steps to the next info packet, given the offset of the previous.

```
: xref          \ -- ; XREF <name>
```

Use in the form XREF <name> to display where <name> is used.

```
: uses          \ -- ; synonym for XREF
```

A synonym for XREF above.

```
: xref-all      \ -- ; cross reference all words
```

Produces a cross reference listing of all the words with cross reference information. This information is often too long to be directly useful, but can be pasted from the console to an editor for sorting, printing, and other post-processing.

```
: xref-unused   \ -- ; cross reference all words
```

Produces a cross reference listing of all the unused words with cross reference information. This information is often too long to be directly useful, but can be pasted from the console to an editor for sorting, printing, and other post-processing.

28.2 Extended String Package

This optional wordset found in */Lib/StringPk.fth* contains the following definitions to aid in the manipulation of counted strings.

`: $variable     \ #chars "name" --`
Create a string buffer with space reserved for #chars characters

`: $constant     \ "name" "text" --`
Create a string constant called "name" and parse the the closing quotes for the content.

`: ($+)          \ c-addr u $dest --`
Add the string described by C-ADDR U to the counted string at $DEST. This word is now in the kernel.

`: $+            \ $addr1 $addr2 --`
Add the counted string $ADDR1 to the counted buffer at $ADDR2. This word is now in the kernel.

`: $left         \ $addr1 n $addr2 --`
Add the leftmost N characters of the counted string at $ADDR1 to the counted buffer at $ADDR2.

`: $mid          \ $addr1 s n $addr2 --`
Add N characters starting at offset S from the counted string at $ADDR1 to the counted buffer at $ADDR.

`: $right        \ $addr1 n $addr2 --`
Add the rightmost N characters of the counted string at $ADDR1 to the counted buffer at $ADDR2.

`: $val          \ $addr -- n1..nn n`
Attempt to convert the counted string at $ADDR1 into a number. The top-most return item indicates the number of CELLS used on stack to store the return result. 0 Indicates the string was not a number, 1 for a single and 2 for a double. $VAL obeys the same rules as NUMBER?.

`: $len          \ $addr -- len`
Return the length of a counted string. Actually performs `C@` and is the same as COUNT NIP.

`: $clr          \ $addr --`
Clear the contents of a counted string. Actually sets its length to zero. Primarily used to reset buffers declared with $VARIABLE.

`: $upc          \ $addr --`
Convert the counted string at $ADDR to uppercase. This acts in place.

`: $compare      \ $addr1 $addr2 -- -1/0/+1`
Compare two counted strings. Performs the same action as the ANS kernel definition COMPARE except that it uses counted strings as input parameters.

`: $<            \ $1 $2 -- flag`
A counted string equivalent to the numeric `<` operator. Uses $COMPARE then generates a well - formed flag.

`: $=            \ $1 $2 -- flag`
A counted string equivalent to the numeric = operator. Uses $COMPARE then generates a well - formed flag.

`: $>            \ $1 $2 -- flag`

A counted string equivalent to the numeric `>` operator. Uses $COMPARE then generates a well - formed flag.

`: $<>            \ $1 $2 -- flag`

A counted string equivalent to the numeric `<>` operator. Uses $COMPARE then generates a well-formed flag.

`: $instr         \ $1 $2 -- false | index true`

Look for an occurance of the counted string $2 within the string $1. If found then the start offset within $1 is returned along with a TRUE flag, otherwise FALSE is returned.

28.3 Extensible CASE Mechanism

A CHAIN is an extensible version of the CASE..OF..ENDOF..ENDCASE mechanism. It is very similar to the `SWITCH` mechanism described in the *Tools and Utilities* chapter.

`: case-chain    \ -- addr ; -- addr`   MPE.0000

Begin initial definition of a chain

`: item:         \ addr n -- addr ;`   MPE.0000

Begin definition of a conditional code block

`: end-chain     \ addr --`   MPE.0000

Flag the end of the current block of additions to a chain

`: in-chain?     \ n addr -- flag ;`   MPE.0000

Return TRUE if N is in the chain beginning at ADDR

`: exec-chain?   \ i*x n addr -- j*x true | n FALSE`   MPE.0000

Run through a given chain using TOS as a selector. If a match is made execute the relevant code block and return TRUE otherwise the initial selector and a FALSE flag is returned.

28.3.1 Using the chain mechanism

```
CASE-CHAIN <foo>
  <n> ITEM: <words> ;
  <m> ITEM: <words> ;
  <k> ITEM: <words> ;
END-CHAIN
```

More items can be added later:

```
<foo>
  <x> ITEM: <words> ;
  ...
END-CHAIN
```

The data structures are as follows:

CASE-CHAIN <foo> generates a variable that points to the last item added to the list.

```
ITEM: generates two cells and a headerless word:
  selector
  link
  headerless word .... exit
```

28.4 Binary Overlays

28.4.1 Introduction

Binary overlays are pieces of the dictionary that have been compiled and saved with relocation information. They can be reloaded as needed and released on demand. Binary overlays are useful when you want to ship tools that are only needed during development, or if you have a large application whose memory footprint you want to reduce by only loading parts of the application when needed.

The binary overlay utility is not part of the kernel, but can be compiled from LIB\OVLVFX.FTH. As of build 3.40.0808, there has been major change in the way overlays are constructed. This change removes many restrictions that were present in earlier builds. To use the new overlay handler, all overlays must be rebuilt.

28.4.2 Using overlays

An overlay is generated by MAKEOVERLAY

```
  MAKEOVERLAY <sourcename> <overlayname>
```

the file <sourcename> is compiled twice. Relocation information is extracted and saved to the overlay along with the raw binary information. If any previously loaded overlays are needed by this overlay, their names are saved in the overlay and they will be automatically reloaded if necessary. After the overlay has been generated, the overlay code is removed. Overlays can be tested by compiling <sourcename> conventionally, and then finally generating the overlay when you are satisfied with it. MAKEOVERLAY preserves and links all vocabularies including SOURCEFILES. Overlay files are saved by MAKEOVERLAY in the current directory. The compiler imposes the following initial condition before the overlay file is compiled:

```
  DECIMAL -SHORT-BRANCHES +SIN +SINDOES
```

MAKEOVERLAY releases all previously loaded overlays. As a consequence, if the overlay to be compiled requires other overlays, you must load them explicitly by specifying them as dependencies before using MAKEOVERLAY. A dependency list is defined by the word [DEPENDENCIES followed by a list of overlay file names as required by LOADOVERLAY below. The list is termininated by DEPENDENCIES]. Use in the form:

```
[dependencies
  primovl secovl ...
dependencies]
makeoverlay MyOvL
```

This will cause MAKEOVERLAY to load the dependent overlays PRIMOVL.OVX and SECOVL.OVX and so on.

When an overlay is reloaded by LOADOVERLAY

```
LOADOVERLAY <overlayname>
```

the binary code and relocation information are loaded. If the overlay file references other overlays, these are loaded before the relocated binary code is installed. Overlay code is loaded into memory allocated from the Windows heap, and are linked in reverse load order, so that the last loaded is found first. The result of this is that the overlays are always loaded in dependency order, and releasing a "leaf" overlay will not affect the dependencies of other previously loaded overlays.

Although overlay files are saved by MAKEOVERLAY in the current directory, LOADOVERLAY will look first in the current directory and then in the directory from which the application was loaded. This allows all overlays and the main executable to reside in the same directory regardless of the current directory, but maintains convenience during development.

An overlay can be released by the use of RELEASEOVERLAY.

```
RELEASEOVERLAY <overlayname>
```

All loaded overlays can be released by RELEASEALLOVERLAYS

```
ReleaseAllOverlays
```

28.4.3 Load and Release actions

A word can be set to excute whenever the overlay is loaded from file or released. These words permit the overlay to allocate and free resources such as memory buffers.

```
' <load-action> SetOvlLoadHook
' <release-action> SetOvlReleaseHook
```

Note that these settings should be in the overlay load file. The stack effect of <load-action> and <release-action> must be neutral, i.e. take nothing and return nothing [–].

28.4.4 File name conventions

From VFX Forth v3.4 onwards, the naming conventions have been changed.

• The internal overlay name is always the output file name after any default extension name has been added by MAKEOVERLAY.
• LOADOVERLAY checks the internal overlay name after adding the default file extension.

The binary overlay files have a ".OVX" extension. The word MAKEOVERLAY creates the overlay for you as follows:

```
MAKEOVERLAY <sourcename> <overlayname>
```

If the source file name does not have an extension, the rules of INCLUDED will be followed, checking for files with extensions ".BLD" ".FTH" ".F" ".CTL" ".SEQ" in that order. If the destination file name does not have an extension ".OVX" will be used. If the destination file name is not provided, the source file name is used with a ".OVX" extension. Thus, just typing MAKEOVERLAY FOO will compile FOO.FTH to create FOO.OVX. The overlay name held by the system is the output file specification as given or created by MAKEOVERLAY, converted to upper case. This is important when reloading the overlay.

If no extension is provided for LOADOVERLAY, a ".OVX" extension will be added to the file name. Thus LOADOVERLAY FOO will check if an overlay called FOO.OVX has been loaded, and will load from file FOO.OVX. Similarly, LOADOVERLAY FOO.OVX will check if an overlay called FOO.OVX has been loaded, and will load from file FOO.OVX.

28.4.5 Version control

Each overlay contains VFX Forth information, and overlays cannot be loaded by a version of VFX Forth other than the one that built it. A user defined version string can be added to the version control information using SETOVLVER, which takes the address of a counted string. The format of the string is entirely user defined, the overlay handler simply checks the strings for identity.

Note that this version of LIB\OVLVFX.FTH requires VFX Forth build 3.40.0808 of 15 March 2002 or later.

28.4.6 Restrictions

The following system state is preserved and restored by the overlay handler.

```
Overlays needed by the current overlay
Vocabularies and vocabulary link
Wordlists and wordlist link
Libraries
Imported functions
```

If you generate other system-wide chains, these will NOT be preserved. To preserve them, modify the code in LIB\OVLVFX.FTH using the xxxIMPORTLINK words as a model. Future versions of this code may support a chain of chains model, but this will require that ALL such chains are anchored in the VFX Forth kernel/application before any overlays are either generated or reloaded.

N.B. If you modify this code, please pass it back to MPE so that it can be incorporated in later builds. This will reduce your maintenance work, our technical support load, and you will benefit from the work of others.

28.4.7 Gotchas

Bad or random data

The overlay is produced by comparing two versions of the binary at different addresses, and generating relocation information from any differences. If a relocation value does not correspond to another overlay or the VFX Forth kernel, the build of the overlay will cause an error. Such errors can be caused by anything that inadvertently changes the data or code generation of the two versions being compared.

Uninitialised buffers

If data space in the dictionary is not initialised at compile time, it may contain random data. Compare:

```
<size> BUFFER: <name>               \ safe
CREATE <name>   <size> ALLOT        \ unsafe
CREATE <name>   <size> ALLOT&ERASE  \ safe
```

Different initial conditions

The initial conditions of directives that affect code generation must be the same for each build. At least the following directives should be considered:

```
+SHORT-BRANCHES  -SHORT-BRANCHES   branch code size
+SIN             -SIN              source inlining
+SIN-DOES        -SIN-DOES         DOES> clause inlining
```

Similarly the starting codition of BASE should also be considered. The compiler imposes the following initial condition before the overlay file is compiled:

```
DECIMAL -SHORT-BRANCHES +SIN +SINDOES
```

Search order issues

When compiling an overlay strict control of the initial search order is often necessary, especially because of redefinitions. We recommend that overlays are constructed from a build file which ensures that other required overlays are installed.

A sign of bad search order control is that the overlay can be correctly built with the source inliner turned off, but will not build with it on.

Long file names

You cannot use file names with spaces, even though GETPATHSPEC is used to input the file names, because the file names are internally used as Forth word names.

Code conflicts with address

There are occasions when a four-byte code sequence matches an address in another overlay, causing false relocation data to be generated. The result will be code that is corrupt after loading.

This situation has been drastically improved by the overhaul of 14 March 2002, but the warning has been left in until we are confident that all situations have been covered.

28.4.8 Overlay glossary

`defer ovl-init-compile  \ -- ; set initial state`

A DEFERred word to set the initial compilation state for both compilations of the overlay source code. The default condition is:

```
  decimal optimised -short-branches +sin +sindoes
```

Do not rely on this word being present in future releases. It is only present for experimental use with very large overlays.

`: [dependencies \ -- ; set up dependency list`

This word is used before MAKEOVERLAY below to define a list of overlays required by the overlay to be made. It is followed by a list of overlay file names as required by LOADOVERLAY below. The list is termininated by DEPENDENCIES]. Use in the form:

```
[dependencies
  primovl secovl ...
dependencies]
```

`: $MakeOverlay  \ c-addr1 u1 c-addr2 u2 --`

Use the first string as the source file name and the second string as the overlay name. This word constructs a MAKEOVERLAY string and EVALUATES it. $MAKEOVERLAY is provided for the construction of higher level overlay management functions.

`: MakeOverlay   \ "src" ["dest"] -- ; MAKEOVERLAY <buildfile> <overlay>`

Creates an overlay by loading an input file, which can itself load other files, and producing an output file. If the source file name does not have an extension, the rules of INCLUDED will be followed, checking for files with extensions ".BLD" ".FTH" ".F" ".CTL" ".SEQ" in that order. If the destination file name does not have an extension ".OVX" will be used. If the destination file name is not provided, the source file name is used with a ".OVX" extension. Thus, just typing MAKEOVERLAY FOO will compile FOO.FTH to create FOO.OVX. The overlay name held by the system is the output specification as given. This is important when reloading the overlay. The compiler imposes the following initial condition before the overlay file is compiled:

```
  DECIMAL -SHORT-BRANCHES +SIN +SINDOES
```

`: SetOvlLoadHook        \ xt -- ; ' <load-action> SETOVLOADHOOK`

This word sets the action to be performed whenever the overlay is loaded from the file. This action is NOT called by LOADOVERLAY if the overlay is already loaded. SETOVLLOADHOOK must be included in the overlay load file.

`: SetOvlReleaseHook     \ xt -- ; ' <release-action> SETOVLRELEASEHOOK`

This word sets the action to be performed when the overlay is released. SETOVLRELEASEHOOK must be included in the overlay load file.

`: SetOvlVer     \ c-addr --`

Sets the address of a counted string added to the version control information. All overlay loads will be checked against this string. SETOVLVER must be used before MAKEOVERLAY. The string can be reset at any time by 0 SETOVLVER.

`: $OvlLoaded?   \ c-addr u -- start true | 0 0`

Converts the string to upper case and tests whether or not the overlay has been loaded, returning its start address in memory and true if loaded, or two zeros if not loaded. See MAKEOVERLAY for a discussion of overlay names.

`: $LoadOverlay  \ c-addr u -- start|ior end|-1`

Uses the given string as an overlay name, and reloads the the overlay if not already loaded. If the overlay name does not have an extension, ".OVX" will be used. Any other required overlays will be loaded before the requested overlay. The start and end+1 address of the overlay code after installation are returned. $LOADOVERLAY is provided for the construction of higher level overlay mangement functions. On error, the start and end values are replaced by ior and -1.

`: LoadOverLay   \ "name" -- ; LOADOVERLAY <name>`

Load an overlay whose name follows in the input stream. See $LOADVERLAY for more details.

`: .overlays     \ -- ; display loaded overlays`

Shows the names of the the loaded overlays.

`: lo            \ "name" -- ; LO <name>`

A synonym for LOADOVERLAY. See $LOADVERLAY for more details.

`: mo            \ "src" ["dest"] -- ; MO <buildfile> <overlay>`

A synonym for MAKEOVERLAY.

`: $ReleaseOverlay       \ c-addr u -- ior`

Release the overlay of the given name, returning a non-zero code if the overlay was not loaded. The name is converted to upper case before the comparison is performed. $RELEASEOVERLAY is provided for the construction of higher level overlay mangement functions. If the overlay was loaded when OVL_IN_DICT was set FALSE (the default), overlays loaded after the specified one will also be removed. If the overlay was loaded when OVL_IN_DICT was set TRUE, the overlay is in the 'kernel' area of the dictionary, and any code compiled or loaded after the overlay will also be removed. Overlays dependent on this one will be removed.

`: ReleaseOverlay        \ "text" -- ; RELEASEOVERLAY <name>`

Uses $RELEASEOVERLAY to release the overlay whose name follows. See $RELEASEOVERLAY for more details.

`: ro            \ "text" -- ; RO <name>`

A synonym for RELEASEOVERLAY. See $RELEASEOVERLAY for more details.

`: ReleaseAllOverlays    \ --`

Releases and unhooks all overlays. Executed automatically by the Exit chain.

`: ovl_in_dict   \ -- addr ; true to load overlays in dictionary ; SFP022`

Set this variable to TRUE to load overlays at the end of the dictionary, rather than in memory allocated from the heap. This is only required in special circumstances. After overlays have been built, restore OVL_IN_DICT to FALSE.

28.5 Configuration files

Application configuration can be done in a number of ways, especially under Windows.

`Registry` A user nightmare to copy from one machine to another

`INI` files Very slow for large configurations (before mpeparser.dll)

`binary` Usually incompatible between versions

`database` Big and often similar to binary

`Forth` Already there, needs changes to interpreter. Independent of operating system.

A solution to this problem is available in *Lib/ConfigTools.fth*. Before compiling the file, ensure that the file GenIO device from *Lib/Genio/FILE.FTH* has been compiled.

The Forth interpreter is already available, but we have to consider how to handle incompatibilities between configuration files and issue versions of applications. The two basic solutions are:

- Abort on error
- Ignore on error

The abort on error solution is already available - it just requires the caller of `included` to provide some additional clean up code.

```
: CfgIncluded   \ caddr len --
  -source-files            \ don't add source file names
  ['] included catch
  if  2drop  endif         \ clean stack on error
  +source-files            \ restore source action
;
```

In VFX Forth, `INTERPRET` is used to process lines of input. `INTERPRET` is `DEFER`red and the default action is `(INTERPRET)`. The maximum line size (including CR/LF) is `FILETIBSZ`, which is currently 512 bytes. If we restrict each configuration unit to one line of source code, we can protect the system by ignoring the line if an error occurs. We also have to introduce the convention in configuration files that actions are performed by the last word on the line (except for any parsing). This action has to be installed and removed, leading to the following code.

```
: CfgInterp     \ --
\ Interprets a line, discarding it on error.
  ['] (interpret) catch
  if  postpone \  endif
;

: CfgIncluded   \ caddr len --
\ Interprets a file, discarding lines with errors.
  -source-files                 \ don't add source file names
  behavior interpret >r
  ['] CfgInterp is interpret
  ['] included catch
  if  2drop  endif              \ clean stack on error
  r> is interpret
  +source-files                 \ restore source action
;
```

28.5.1 Loading and saving configuration files

`: CfgInterp     \ --`

A protected version of `(INTERPRET)` which discards any line that causes an error.

`: CfgIncluded   \ caddr len --`

A protected version of `INCLUDED` which discards any line that causes an error, and carries on through the source file.

`: [SaveConfig   \ caddr len -- struct|0`

Starts saving a configuration file. Creates a configuration file and allocates required resources, returning a structure on success or zero on error. On success, the returned *struct* contains the `sid` for the file at the start of *struct*.

`: SaveConfig]   \ struct --`

Ends saving a file device by closing the file, releasing resources and restoring the previous output device.

`: SaveConfig    \ caddr len xt --`

Save the configuration file, using *xt* to generate the text using `TYPE` and friends. The word defined by *xt* must have no stack effect.

28.5.2 Loading and saving data

We chose to support five type of configuration data:

- Single integers at given addresses. This copes with `variable`s directly and `value`s with `addr`.
- Double integers at given addresses.
- Counted strings
- Zero terminated strings
- Memory blocks.

All numeric output is done in hexadecimal to save space, and to avoid problems with `BASE` overrides. All words which generate configuration information **must** be used in colon definitions.

`: \Emit         \ char --`

Output a printable character in its escaped form.

`: \Type         \ caddr len --`

Output a printable string in its escaped form.

```
: .cfg$         \ caddr len --
```

Output a string in its escaped form, characters in the escape table being converted to their escaped form. The string is output as Forth source text, e.g.

```
s\" escaped text\n\n"
```

```
: .sint         \ x --
```

Output x as a hex number with a leading '$' and a trailing space, e.g.

```
$1234:ABCD
```

Single Integers

Single integers are saved by `.SintVar` and `.SintVal`.

```
' (SintVar) SimpleCfg: .SIntVar \ "<name>" --
```

Saves a single integer as a string. `<name>` must be a Forth word that returns a valid address. Generates

```
$abcd <name> !
```

Use in the form:

```
.SIntVar MyVar
```

```
' (SintVal) SimpleCfg: .SIntVal \ "<name>" --
```

Saves a `VALUE` called `<name>`. Generates

```
$abcd to <name>
```

Use in the form:

```
.SIntVal MyVal
```

Double Integers

Double integers are saved by `.DintVar`.

```
' (DintVar) SimpleCfg: .DIntVar \ "<name>" --
```

Saves a double integer as a string. `<name>` must be a Forth word that returns a valid address. Generates

```
$01234 $abcd <name> 2!
```

Use in the form:

```
.SIntVar MyVar
```

Counted strings

Counted strings are saved by `.C$CFG`.

```
' (c$cfg) SimpleCfg: .C$var     \ "<name>" --
```

Saves a string `<name>` must be a Forth word that returns a valid address. Generates

```
s\" <text>" <name> place
```

Use in the form:

```
.C$Var MyCstring
```

Zero terminated strings

Zero terminated strings are saved by `.Z$var`.

```
' (z$cfg) SimpleCfg: .Z$var     \ "<name>" --
```

Saves a zero terminated string at `<name>` which must be a Forth word that returns a valid address. The output consists of one or more lines of source code, following lines being appended to the first.

```
s\" <text>" <name> zplace
s\" <more text>" <name> zAppend
...
```

Use in the form:

```
.Z$var MyZstring
```

Memory blocks

Memory blocks are output by

```
.Mem <name> len
```

`<Name>` must be a Forth word that returns a valid address. `Len` must be a constant or a number. The output takes one of three forms, depending on `len`.

```
bmem <name> num  $ab $cd ...
wmem <name> num  $abcd $1234 ...
lmem <name> num  $1234:5678 $90ab:cdef ...
```

A block of memory is output by

```
.Mem <name> len
```

`<Name>` must be a Forth word that returns a valid address. `Len` must be a constant or a number.

```
: BMEM          \ "<name>" "len" --
```

Imports a memory block output in byte units by `.Mem`.

```
: WMEM          \ "<name>" "len" --
```

Imports a memory block output in word (2 byte) units by `.Mem`.

```
: LMEM          \ "<name>" "len" --
```

Imports a memory block output in cell (4 byte) units by `.Mem`.

29 ClassVfx OOP

There are two sets of documentation for the *ClassVFX* system. There is a chapter in the main VFX Forth manual, and there is a full PDF manual in the *Manual* subdirectory of *Lib\OOP\ClassVFX*.

29.1 Introduction

The source code is in the directory *Lib\OOP\ClassVFX*. The file *MakeClassVfx.bld* is compiled to produce the production version of the code. *TestClassVfx.fth* contains test code.

ClassVFX was developed over a number of years in collaboration with Construction Computer Software of Cape Town, South Africa. We gratefully acknowledge their collaboration and permission to release it. ClassVFX is heavily used in their construction industry planning software, which is one of the largest Forth applications ever written. Modifications to ClassVFX will only be released after the agreement of CCS.

ClassVFX is a halfway house between a full object oriented system and an intelligent structures system. Types, or classes, can be defined with single inheritance. Method names have to be predefined using

```
OPERATOR: <method-name>
```

Field, or data member, names are private, but are accessible using a dot notation. There are no equivalents of `SUPER` and `SELF`. There is no late binding.

In this documentation **type**s and **class**es are synonymous. **Object**s are **instance**s of a **type**. Objects have a default action if no method is specified. Usually the default action is to fetch the contents of the object, but in a few cases the default action is to return an address.

Types/Classes can have both class and instance methods. The default method for a type is to create an instance. If a type is used inside a colon definition a local variable version is created and destroyed at run time.

Operators, or methods, must be declared as above before use.

29.2 How to use TYPE: words

`TYPE:` definitions may be used in four ways:

- as an abstract template which is used with a base address on the stack. In this case `Point` is a type (class).

  ```
  Point.x or x
  ```

- to define an instance of a structure in the dictionary, e.g.

  ```
  Point: MyPoint
  ```

- to define a local variable inside a colon definition, but outside any other local variable defining mechanism. If another locals defining mechanism such as the ANS `LOCALS| ... |` mechanism or the MPE `{ ... }` mechanism has been used the use of `Point: foo` inside a colon definition will simply add `FOO` to the existing local frame.
- to define a field inside another `TYPE:` definition.

```
operator: <method1>
operator: <method2>
operator: <method3>

type: line:
  point: start
  point: end

  :m <method1>     ... ;m
  :m <method2>     ... ;m
  mruns <method3>  <some-word>
  :m <xxx>         a b c d  ;m  structure-method

end-type

Line: MyLine
  1 2 to Myline.start
  5 to Myline.end.y
```

At runtime, the method operates on the address of the data. Because of this, a method which requires the address of the instance structure has to be marked by the word `STRUCTURE-METHOD` which causes the compiler to generate the address of the instance structure, **not** the type structure.

ClassVFX allows both `CLASS` and `INST`ance methods to be defined for a type. `INST`ance methods, the default, operate on the address of the data item. `CLASS` methods operate on the address of the type data structure. As described above, `STRUCTURE-METHOD`s operate on the instance data structure.

Single inheritance can be defined using `SUPERCLASS <type>` or `INHERITS <type>` before any field or method is defined.

```
TYPE: <type>  SUPERCLASS <supertype>
  ...
END-TYPE
```

At run-time, methods are provided with the address of the required data. CLASS/TYPE methods receive the address of the TYPE/CLASS data structure, INSTance methods receive the address of the data item. INSTance methods that require the address of the instance data structure must be marked by `STRUCTURE-METHOD`. Methods may be defined as nameless words:

```
  :M <method-name> ...  ;M
```

or as the action of a method:

```
  MRUNS <method-name> <action-name>
```

The code below is taken from the definition of the default type.

```
class     \ define methods for the type
   :m default    make-inst  ;m
   :m sizeof     type-size @ ?complit  ;m
   :m addr       ?complit  ;m
inst      \ define methods for the instance
   mruns default noop
   :m sizeof     type-size @ ?complit  ;m  structure-method
   mruns addr    noop
   :m offsetof   off-start @ ?complit  ;m  structure-method
   :m +offsetof  off-start @ ?complit+  ;m  structure-method
```

To use the nested field system, the Forth system has been modified to accept compound names in which the elements of the structure are separated by the '.' character. This feature is enabled and disabled by the words `+STRUCTURES` and `-STRUCTURES`.

29.3 Predefined types

```
char:         byte - 8 bit variable
word:         word - 16 bit variable
int:          long - 32 bit variable and synonyms
  dword:
  long:
  ptr:
xlong:        longlong - 64 bit variable
bytes:        byte array: size specified by n BYTES BYTES:
cstring:      counted string: size specified by BYTES before CSTRING:
zstring:      zero term. string: size specified by BYTES before ZSTRING:
field:        byte array, only ADDR operator, size specified by BYTES
```

29.4 Predefined methods/operators

Note that not all predefined types support all methods.

```
 0 operator default           usually a fetch operation
 1 operator ->                store operator
 1 operator to                  "
 2 operator addr              address operator
 3 operator inc               increment by one
 4 operator dec               decrement by one
 5 operator add               n add to
 6 operator zero              set to 0
 7 operator sub               subtract from
 8 operator sizeof            size
 9 operator set               set to -1
10 operator offsetof          offset in object
11 operator +offsetof         add offset in object
12 OPERATOR FETCH             get contents
13 OPERATOR ADDR\CNT          address under count
14 OPERATOR TWIST             change endian of the data type
15 OPERATOR CONSTRUCT         build an instance of this type
15 OPERATOR MAKE              build an instance of this type
op# ADDR OPERATOR ADDROF
OPERATOR: <=>    type_addr_y <=> <type_x> --- set typedef_x = typedef_y
  op# <=> OPERATOR <copy>
OPERATOR: <blank>             blank object for object size
OPERATOR: <erase>             fill obj with null for object size
OPERATOR: <COUNT>
OPERATOR: <make>
OPERATOR: <destroy>
OPERATOR: <INIT>
OPERATOR: <fetch>
```

29.5 Example structure

```
TYPE: POINT:  \ --
\ Defines a type called POINT: with the following fields )
  PROVIDER: NOOP        \ defines the address provider, defaults to NOOP
  0 OFFSET:             \ defines the initial offset, defaults to 0
  INT: Y
  INT: X
  10 BYTES FIELD: FOO
                        \ fetch operation
  :m default
     2@
  ;m
  mruns to       2!
  ...

END-TYPE
```

29.6 Data structures created by TYPE:

`TYPE:` definitions, fields, objects and so on all use a common data structure that is generated by the defining words.

These structures are associated with a word (the address provider) that can provide the starting address of the structure implementation. By supplying the cfa of `NOOP`, no address is provided, and so the structure is purely a template. For templates, address provider = 0 or `NOOP`, an offset may also be defined. `NOOP` and 0 are the default address provider and offset of templates.)

A similar structure is used for instances of a `TYPE:`. These are created by the word `MAKE-INST`.

29.6.1 TYPE: definitions

The following structure is created by `TYPE:`

```
   header                          standard PFW layout
0  jmp do_type                     5 bytes
1  cfa of address provider         4 bytes
2  initial offset                  4 bytes  0 for class
3  link to last field defined      4 bytes
4  type size - final offset        4 bytes
5  Magic number                    4 bytes
6  anchor of instance method chain 4 bytes
7  anchor of type method chain     4 bytes
8  link to previous type defined   4 bytes
9  private wordlist                ? bytes
```

29.6.2 MAKE-INST definitions

The following structure is created by `MAKE-INST`

```
   header                          standard PFW layout
0  jmp do_inst                     5 bytes
1  cfa of address provider         4 bytes
2  offset from start of type       4 bytes
3  link to last instance of type   4 bytes
4  size of instance data           4 bytes
5  0                               4 bytes
6  pointer to TYPE/CLASS           4 bytes
7  data if static
```

29.7 Local variable instances

When an instance is defined inside a colon definition, an uninitialised local variable/array is built. Several instances can be built. Normally the size of all local variables is rounded up to a cell boundary by the compiler

29.8 Defining methods

`: :M            \ struct -- struct  ; :M <operator> <actions ...> ;M`
defines the start of a method. The method/operator name must follow.

`: ;M            \ struct -- struct ; SFP012`
marks the end of a method definition.

```
: MRUNS         \ struct -- struct  ; MRUNS <operator> <actions> ;M
```
Defines a method which runs a previously defined word.

29.9 Create Instance of an object

```
: CREATE-INST   \ "<name>" -- ; -- addr
```
From VFX Forth v4.4, this is a synonym for `CREATE`. When compiling on previous VFX versions instances needed to be immediate.

```
: make-inst     \ class -- ; i*x -- j*y ; build instance of type
```
Builds an instance of a `TYPE:`. This word has serious carnal knowledge of the internal workings of VFX Forth. Don't call us for help!

29.10 Defining TYPE: and friends

```
create type-template    \ -- addr
```
The type chain from which others are derived.

```
CREATE ptr-template     \ -- addr
```
The ptr chain from which others are derived.

29.10.1 TYPE definition

```
: type:-runtime \ type-struct --
```
The run-time action of children of `TYPE:`.

```
: CURR-TYPE-SIZE        \ -- u
```
Use between TYPE: <name> and END-TYPE to return the current size of the type.

```
: TypeChildComp,        \ xt --
```
Compile a child of `TYPE:`.

```
: type:         \ -- struct ; --
```
Start a new `TYPE:` definition.

```
: PTR:          \ -- struct ; --
```
Make a new structure defining word.

```
: end-type      \ struct --
```
Finish off a `TYPE:` definition

```
: EXTEND-TYPE   \ "<type>" -- struct ; EXTEND-TYPE <type> ... END-TYPE
```
Extend the given `TYPE:` definition.

```
: SUPERCLASS    \ struct "<type>" -- struct
```
Use this inside a `TYPE:` definition before defining any data or methods. The current type will inherit the data and methods of the superclass.

```
: INHERITS      \ struct "type" -- struct
```
A synonym for `SUPERCLASS`.

```
: provider:     \ struct "name" -- struct ; <name> is address provider
```
Sets a different address provider.

```
: with:         \ -- ; WITH: <some-provider> LINE: <myline>
```
Used before declaring an instance to override the default address provider.

```
: SKIPPED       \ struct size -- struct
```
Increase overall size of struct by size. `SKIPPED` can be used to jump over items from a previous instance.

`: OFFSET:        \ struct offset -- struct`
Define the offset of a `TYPE:` as starting at a value other than zero. Must be used before any data is defined.

`: TypeCast:     \ -- ; TYPECAST: <inst> <type>`
Forces a previously defined instance to be a pointer to a type/class.

`SYNONYM PointsTo: TypeCast:     \ -- ; synonym for TYPECAST:`
Forces a previously defined instance to be a pointer to a type/class.

`: type-self     \ -- type`
Used in TYPE: <name> ... END-TYPE to refer to the type/class being defined.

`: EXECUTE-MEMBER-METHOD \ struct-inst member-inst methodid ---`
Attempt to execute method for inst. Return true if successful.

`: EXECUTE-PTR-MEMBER-METHOD     \ member-inst methodid ---`
Attempt to execute method for inst. Return true if successful.

`: EXECUTE-MEMBERS       \ inst method --`
Apply the given method to all members of the instance of a type/class.

`: TWIST-STRUCTURE       \ inst --`
Twist structure method

`: INIT-STRUCTURE        \ inst --`
Init structure method

29.11 Dot notation parser

In order to deal with structures and fields without having to backtrack the input stream or the execution order, an additional stage is added to the Forth parser to allow phrases of the forms:

```
inst.field
inst.field.field
type.field
type.field.field
```

to be parsed, where each item is separated by a dot character. The first item must be an instance of a type or a type. If it is an instance, the address is provided, otherwise the base address is assumed to be on the stack. Any items between the first and last item add their offsets to the address, and the last item performs the usual operation of the field as defined by an operator. For example:

```
type: point:
  int: x0
  int: y0

  :m <op1> ... ;m
  :m <op2> ... ;m
end-type

point: Mypoint
  5 to MyPoint.x0
```

We might define a line as joining two points:

```
type: line:
  point: p1
  point: p2
  ...
end-type

line: MyLine
  5 to MyLine.p2.y0
```

`: +structures   runword \ --`
Switch on the structure compiler.

`: -structures   runword \ --`
Switch off the structure compiler.

30 CIAO - C Inspired Active Objects

CIAO is an OOP package modelled on C++ for VFX Forth. CIAO is designed to provide easy interfacing to host operating system structures that are built around a C++ model.

The source code for CIAO is in the *Lib\oop\Ciao* directory, as are several example class files. To rebuild CIAO, compile the file *ciao.bld*.

30.1 Token and Parsing Helpers

Various utilities and factors useful for parsing text.

`buffer: token-buffer`

A Memory buffer used to hold the result of the last token parse. The size of this buffer comes from the environement variable MAX-CHAR and is MAX-CHAR + 1 characters in length since the string is stored as a counted string.

`: new-word      \ char -- $`

This is a replacement for WORD which places the output in the token buffer.

`: peek-token    \ -- c-addr u`

Copy the next token into the TOKEN-BUFFER without permanent change to the input specification (uses SAVE-INPUT and RESTORE-INPUT). Returns TOKEN-BUFFER as a c-addr u pair.

`: drop-token    \ --`

Throw away the next token *without* corrupting TOKEN-BUFFER.

`: ciao-token    \ -- c-addr u`

Grab the next space delimited token and return c-addr u. Fills TOKEN-BUFFER.

`: bracketed?    \ c-addr u -- flag`

Is the string C-ADDR U bracketed?

30.2 The THIS Stack

The heart of this OOP implementation is the concept of "THIS". Just like C++ "THIS" returns the currently active object instance pointer. Instance data is accessed via this pointer as are the "virtual" methods.

THIS is kept in a form of stack.

`: >this         \ val --`

Set THIS to VAL. (Preservation is taken care of in the compiler.)

`: this          \ -- instance-pointer`

Return the current instance pointer. Only valid within a method declaration.

30.3 CIAO Constants and Internal Data Stores

`SCOPE_PUBLIC Value CurrentScope \ -- n`

When defining a derived class this holds the scoping type employed.

`0 Value CurrentClass    \ -- n`

When defining a class this points to its CLASS structure.

`0 Value DefFlags        \ -- n`
The declaration flags to be employed by the next method or data member defined in the current class. Records information from control definitions such as VIRTUAL and STATIC.

`0 Value CurrentDefClass \ -- class`
During compilation of a code method, this value holds a pointer to the associated CLASS structure.

`0 Value CurrentDefXT    \ -- xt`
During compilation of a code method, this value holds a pointer to the XT of the definition. See the CIAO-COLON hook for details.

`0 Value CurrentDefList  \ -- list`
During compilation of a code method, this value holds a pointer to the internal method list to be used. The list will either be the classes public, protected or private chain depending on the scope at the time of the method declaration prototype.

`variable class-base-mem \ -- addr`
This variable holds the value of HERE after the building of CIAO. It is used to sanity check the values passed to the instance destruction definition DELETE. Any passed value between this variables value and the current HERE is in dictionary space and must be a static instance which cannot be DELETEd.

30.4 Search Order Utilities

`: NSEARCH-WORDLIST      \ WIDN .. WID1 N C-ADDR U -- XT FLAG | 0`
A most useful definition. FIND takes a counted string but searches the whole search-order. SEARCH-WORDLIST takes a C-ADDR U pair but only searches one wordlist. This definition combines the two, and looks through a number of wordlists for a name described by a C-ADDR U pair. Usually used in association with GET-ORDER to provide a more useful version of FIND.

`: (FindClass)   \ c-addr u -- ptr | THROW`
Run through the current search-order lookin for the name supplied. If the name is found then a >BODY @ is employed on the XT to look for the MAGIC_CLASS identifier. Never called directly, this definition is run from FINDCLASS via CATCH to protect against the times when the token is found but is not a class. Due to the exception handling abilities of CATCH under VFX, this operation should be safe no matter what XT it is employed against.

`: FindClass     \ c-addr u -- ptr | ABORTs`
Invoke (FINDCLASS) via CATCH. Will look for the token supplied and if found will ensure it really is a class definition. ABORTs with text if anything goes wrong.

30.5 Method Lists

The Method Lists hold all the required compiler information for each method within a class. In CIAO, methods don't ever actually exist as regular Forth words. Instead the act of defining a class builds the method lists. Each class has three of these, one for each valid scope (public/protected and private).

30.5.1 The Format of a Method List.

```
| Link | Type | Param1 | Param2 | Name Len | Name Text |
| CELL | CELL |  CELL  |  CELL  |   CHAR   |  n chars  |
```

`Link` Pointer to start of previous list entry (or 0 for top)

`Type` The type of the method (see types below)

`Param1` Parameter 1, varies depending on TYPE.

`Param2` Parameter 2, varies depending on TYPE.

`NameLen` Length of method name.

`NameText` The text for the method name.

30.5.2 TYPE_DATA

Describes an instance data buffer, PARAM1 is the base offset from THIS.

30.5.3 TYPE_STATICDATA

Describes a static data buffer. A static data buffer is placed within the global dictionary rather than being offset from THIS. The net result is that all instances of the owning class and any derived classes share the same location for this data element. PARAM1 is the address in global space of the buffer start.

30.5.4 TYPE_CODE

The default code method type. PARAM1 points to a CELL in global dataspace which will contain the XT of the method body as soon as it becomes available. A CODE method cannot be re-defined or rewritten and it's behaviour is inherited by any derived class.

30.5.5 TYPE_STATICCODE

The second type of code method. It behaves in a similar fashion to TYPE_CODE except the instance pointer THIS is not valid within the method body. These means that a static member has no access to any other member of the class which is non-static. A static member can also be invoked from a colon definition or the interpreter by using the "named scope override" syntax, which does not require an instance pointer. Primarily used to store "normal" functions in a restricted namespace. Ie "do <something> in the name of <some class>"

30.5.6 TYPE_VIRTUALCODE

One of the most useful syntactic additions to C++ was the virtual method. A virtual method can best be described as a method in a base class which you expect to have to modify or replace in a derived class. PARAM1 holds a 0 based index into a table of XTs called a "vtable". Each class has a vtable which in the case of a derived class is initially inherited from the superclass. A derived class can either omit its function body (and thus inherit the behaviour of the superclass) or it can define its own body which can also optionally elect to invoke the superclass's body by using the named scope override syntax. Therefore a virtual method can be either modified or replaced within the context of a derived class. A particularly useful feature of the usefulness of virtual methods can be seen later.

30.5.7 TYPE_CLASS

This type of method specifies a static instance of another class as being a part of the current. PARAM1 specifies the class type whilst PARAM2 specifies the offset from THIS for the instance pointer of the contained class.

30.5.8 TYPE_CLASSPTR

A special form of data store which holds a pointer to a class instance. PARAM1 specifies the class type and PARAM2 the offset from THIS to a single cell. This cell will hold an instance pointer which can be dynamically assigned.

30.5.9 The definitions which deal with lists are:

`: list_link     \ *entry -- *link`
Modify a pointer to a list head to point to the link field.

`: list_type     \ *entry -- *type`
Modify a pointer to a list head to point to the type field.

`: list_param1   \ *entry -- *param1`
Modify a pointer to a list head to point to the param1 field.

`: list_param2   \ *entry -- *param2`
Modify a pointer to a list head to point to the param2 field.

`: list_namelen  \ *entry -- *namelen`
Modify a pointer to a list head to point to the namelen field.

`: list_name     \ *entry -- *name`
Modify a pointer to a list head to point to the name field.

`: .list-type    \ n --`
Given contents of a list entry's type field print it's name as an ascii text string.

`: .list-entry   \ *entry --`
Supplied with a pointer to a list entry this definition will print its contents in human readable form.

`: .list         \ *head --`
Supplied with the address of a variable which points to a list entry this definition will walk backwards through the linked list performing .LIST-ENTRY on each in turn.

`: +LIST         \ Type Param1 Param2 c-addr u *list-head --`
Using the first 5 parameters lay a list-entry structure in the dictionary and add it to the end of the list whose anchor address is at the address pointed to by *LIST-HEAD.

30.6 Operator List

Each class has a linked list called the Operator Chain. This list contains the mapping of operator-id number against class method list entry (from above).

An operator structure entry consists of three fields, the link, the operator id number and a method-list pointer.

`: oplist_link             ;`
Given a pointer to an operator structure return pointer to link

`: oplist_op#    1 cells + ;`
Given a pointer to an operator structure return pointer to op#

`: oplist_list   2 cells + ;`
Given a pointer to an operator structure return pointer to *list

`: .op#                  \ n --`

Where possible print human readable description for operator id N

`: .oplist-entry \ *entry`
Display an operator structure in human readable form.

`: .oplist       \ *head --`
Given a pointer to the head of an operator chain from a class, call .OPLIST-ENTRY for each member.

`: +OPLIST       \ op# *list-entry *head --`
Add record to the operator chain anchored at *HEAD

30.7 The CLASS structure

All classes defined have the same structure:

```
Size      Navigation Word        Useage
-------------------------------------------------------------------
CELL      class_magic            A magic 32 bit number used
                                 to signify a CLASS structure.
CELL      class_super            Pointer to parent class for
                                 a derived class object.
CELL      class_private          Method list anchor for PRIVATE
                                 definitions.
CELL      class_protected        Method list anchor for PROTECTED
                                 definitions.
CELL      class_public           Method list anchor for PUBLIC
                                 definitions.
CELL      class_opchain          Anchor for the operator chain
                                 for this class.
CELL      class_sizeidata        Size of Instance data required.
CELL      class_#vtable          Number of entries in the
                                 virtual method table.
CELL      class_pvtable          Pointer to the virtual method
                                 table.
```

`: .class       \ "name" --`
Display as much information about the class "name" as possible in a human readable form.

30.8 Method Searching

Definitions used to find a given method within a class.

`: FindMethodInClass     \ c-addr u *class -- ptr SCOPE | -1`
Given a string containing the method name and a pointer to a class structure, this definition attempts to get the method list entry for that method. On success a pointer to the method list structure is returned as well as the SCOPE indicator, if the method does not exist in the specified class a -1 is returned.

30.9 Default Method Actions

Any code method has a default action assigned when it is prototyped as a debugging aid. Invoking a method for which you have defined no code will give a polite message via ABORT"

```
: vcrash        \ ?? --
```
Default action for prototyped virtual methods.

```
: scrash        \ ?? --
```
Default action for prototyped static methods.

```
: icrash        \ ?? --
```
Default action for prototyped instance methods.

30.10 Method Scope Specification

During class definition the scope can be altered. These definitions are used to control/handle scoping.

```
: public:       \ --
```
During CLASS definition set the current scope to public.

```
: protected:    \ --
```
During CLASS definition set the current scope to protected.

```
: private:      \ --
```
During CLASS definition set the current scope to private.

```
: GetCurrentList        \ -- *list-head
```
Return the method list pointer for the current scope.

30.11 Name Format Checking

In order to reserve characters to provide the syntax for typecasts, scope overrides and method definition certain characters are illegal for method and class names.

Brackets are illegal, since they are used to perform typecasts.

Colon is an illegal character since it is used for scope overrides.

Period (dot) is illegal since it is used for compound invokations.

Star (*) is illegal since it declares an instance pointer.

```
: ?validname    \ c-addr u --
```
Check the name string supplied is valid for either a class or name. Causes an ABORT" on failiure.

30.12 Method Type Overrides

Methods and instance variables defined in a class can have various attributes, these are controlled by simple indicator words.

```
: static        \ --
```
Modify the global DEFFLAGS to include the static type.

```
: virtual       \ --
```
Modify the global DEFFLAGS to include the virtual type.

```
: post-def      \ --
```

Clear the global DEFFLAGS, called after a member definition to reset ready for the next member.

30.13 Data Method Prototyping

These routines are used within a CLASS or STRUCT{ definition to define data members.

`: buff:          \ size "name" --`

Define a data member called "name" of SIZE bytes. By default instance specific data is created, if the member was modified by the STATIC keyword, then global space is allocated. STATIC data members share the same memory location for all instances of the class and any derived classes.

`: cell:          \ "name" --`

A shortcut for a BUFF: of one cell.

`: char:          \ "name" --`

A shortcut for a BUFF: of one char.

30.14 Code Method Prototyping

These routines are used within a CLASS or STRUCT{ definition to define methods.

`: static-meth:  \ "name" --`

The action invoked by METH: when the STATIC modifier was present. Static members have no access to THIS or instance data and like static data members are shared between all instances of the owning class and any derived classes.

`: virtual-meth: \ "name" --`

The action invoked by METH: when the VIRTUAL modifier is in force. Virtual methods can be given a code definition for a class and later modified in a derived class.

`: instance-meth:        \ "name" --`

The default action of METH: creates a method associated with that class.

`: meth:          \ "name" --`

Create a code member (method). Dispatches to one of the above definitions depending on any applied modifiers.

30.15 Class Method Prototyping

These routines are used within a CLASS or STRUCT{ definition to define members which are in turn classes.

`: inst:          \ *class "name" --`

Embed an instance of the supplied class under the given "name".

`: iptr:          \ *class "name" --`

Create a typed pointer for the given class inside the current one. NOT IMPLEMENTED YET!

30.16 Operator Association

This code is used to associate an operator with a given method in a class or structure.

`: FindClassOperator     \ op# *class -- *list-entry true | false`

Given an operator ID and a class pointer attempt to locate the method list entry associated with that id.

```
: AddOperatorToClass    \ op# *list-entry *class --
```

Routine used to bind a method-list entry to an operator id for the given class.

```
: oper:         \ "name" --
```

Attempt to assign the method "name" as the action of the currently active operator in the current class.

30.17 CLASS Definition

Code to create CLASS definitions.

```
: derived?      \ "text" -- true | "" -- false
```

A look-ahead parsing definition used as a factor in CLASS to see if the class name is followed by a " : [<scope>] <name> " string for defining derived classes.

```
: derived-scope \ c-addr u -- scope flag
```

Another parsing definition used by CLASS, after passing the DERIVED? test the next token is checked for a scope setting. If the next token is one of "public, protected or private" then the scope id is returned and a true flag indicating the next token has been used, otherwise SCOPE_PUBLIC is assumed and a false return flag tells CLASS that this token is the actual base class name.

```
: class         \ "name [ : [ <scope> ] <super> ]" -- ; Exec: -- ptr
```

Begin a new class definition called NAME. If the new class is derived from a base class, the method lists are copied depending on C++ scoping rules, as is the Instance data size, operator chain and vtable size. When a CLASS definition is invoked it can do one of two things: If invoked within another CLASS definition it will invoke the class member instance creation (See previous section "Class Method Prototyping"), at any other time a pointer to the class structure is returned.

```
: end-class     \ --
```

Finish the definition of the current CLASS. If this is a derived class, the vtable from the parent is copied. After that any new vtable entries have the default crash vector attached.

30.18 STRUCTures - A new slant on CLASS

Under CIAO (like C++) a structure is a class. The only technical difference is that by default, members of a STRUCT{ are public.

STRUCT{ is provided for more syntactic reasons than technical. It is expected that STRUCT{ is used to declare structures consisting entirely of public data mapped in a contiguous manner so it can be be used with operating system supplied structure pointers.

```
: struct{       \ "name [ : <scope> <super>" -- ; Exec: -- ptr
```

Begin a new STRUCTure definition. Used in the same fashion as CLASS.

```
: }             \ --
```

Terminate a STRUCTure definition. See also END-CLASS.

30.19 Colon and SemiColon Override

CIAO overrides the standard Forth : and ; definitions to allow for method definitions. After a class has been defined it is necessary to write the actual code for any methods prototyped within it. CIAO like C++ takes a method name as being in the form

```
<class>::<method>
```

`: ciao-colon    \ "name" -- | "name" -- "name"`

The new action of : when CIAO is installed. Pre-parses the name to see if it contains a double colon. If not then the original Forth : is called. If a double colon is found the assumption is that this definition is a method. The first portion (before the ::) is taken to be a class name and the second portion the member name. The class is looked up and its *class pointer stored in a global, then the method name is looked up within that class and it's method list entry is also stored. Compilation is then triggered by :NONAME and the XT of this definition kept for use in ;

`: ciao-semicolon        \ --`

The CIAO over-riding action of ; to handle the closing of method definitions. If the current definition was not a method-def then only the original ; is invoked. Otherwise ; is invoked and the XT of the method is patched into the class structure depending on the method type.

`: :             \ "name" --`

The actual overload of Forth's :

`: ;             \ --`

The actual overload of the base Forth ;

30.20 OOP Compiler/Interpreter Extension Core Part 1 - EVALUATE BUFFER

This code is used by the method compiler extension in the Forth interpreter loop. It is a number of definitions used to create strings to pass to EVALUATE.

`1024 buffer: CompileBuffer`

The buffer used to build strings for EVALUATE.

`: ResetCompileBuffer    \ --`

Resets the COMPILEBUFFER for a new string.

`: ciao-evaluate \ c-addr u --`

All EVALUATEs for CIAO come through here.

`: EvaluateCompileBuffer \ --`

Pass the COMPILEBUFFER string to CIAO-EVALUATE.

`: $+RCB          \ c-addr u --`

Append the string in C-ADDR U to the COMPILEBUFFER.

`: n+RCB          \ n --`

Append the ascii representation of the number N to the COMPILEBUFFER. The string added is in the form "$<hex> "

30.21 OOP Compiler/Interpreter Extension Core Part 2 - Method Compile

These definitions handle the compilation of method-list entries depending on type.

`: CompileMethod_DATA \ *list-entry --`
Compiles code for an Instance Data member. Generates a pointer by laying code for "THIS <offset> +".

`: CompileMethod_STATICDATA \ *list-entry --`
Compiles code for a static data member. This is simply a literal address of the global space.

`: CompileMethod_CODE \ *list-entry --`
Compiles code for an instance code member. If the member has already been bound to a definition then the definition XT is compiled along with execute (i.e. " $<XT> execute " is compiled) otherwise a pointer to where the XT will be stored is compiled with a @ execute.

`: CompileMethod_STATICCODE \ *list-entry --`
Compiles code for a static code member. This is a literal address of where the XT will be and a fetch-execute.

`: CompileMethod_VIRTUALCODE \ *list-entry --`
Compiles code for a virtual method. It compiles the following code.

```
$<virtual-method-index>          \ the index into the vtable
cells                            \ convert to vtable offset
this                             \ Get current instance pointer
cell- @                          \ Fetch objects vtable pointer
+ @                              \ extract XT from vtable
execute                          \ and run it
```

`: CompileMethod_CLASS \ *list-entry --`
Compile code for a class instance member. This is almost identical to instance data.

`: OperatorProcess \ *class --`
Compile code for any current operatortype.

`: MethodTokenCompileFromList \ *entry --`
The global factor for this section. Given a method-list-entry it will dispatch to one of the COMPILEMETHOD_xxx definitions depending on type.

30.22 OOP Compiler/Interpreter Extension Core Part 3 - Single Token Check

The single-token check is used at the beginning of the Forths token interpret before the normal FIND. This is used to provide some special overrides to single Forth tokens. Namely:

1. If defining a code method, other members of the current class can be invoked simply by name. Therefore when defining a method any single token needs to be looked up in the method table before checking the normal Forth dictionary.

2. A token can be preceeded by :: which enforces that the name is searched for in global name space (the Forth dictionary) regardless. This is usually used to get you out of rule #1. If for instance you are defining a method for a class which has a member called DUMP, simply entering "DUMP" will compile a reference to that member, if you actually want to use the Forth DUMP you would type "::DUMP"

3. A token can consist of a class-name and method name separated by a double-colon. NOT IMPLEMENTED YET!. This should compile a reference to a STATIC member of a class.

`: single-token  \ c-addr u -- flag`

Does the single-token-check operation and returns TRUE if the token has been processed. If the token should be passed on to the normal Forth lookup FALSE is returned.

30.23 OOP Compiler/Interpreter Extension Core Part 4 - Compounds

`: Process1stToken       \ c-addr u -- ap 0 | code-to-throw`

Used to handle the first part of a dot-notation compound. The first 'token' needs to ultimately lay the code generate THIS and needs to locate the correct class structure pointer that begins this invocation (called the address-provider). How the first token is actually translated depends on a number of rules:

1. If defining a method, the first token may be a class or class pointer member of the current class.

2. If the first token is surrounded by brackets it's a typecast. the name in brackets identifies the address provider whilst the instance pointer is assumed to already be top of the data stack.

3. Next it could be that the first token is a named scope override. (as in rule 3 for single-token) This behaviour is NOT IMPLEMENTED YET! This is generally used for a virtual method in a derived class to invoke the action of the virtual method associated with it's parent.

4. Finally the token can either be a name in global space or a name defined as a LOCAL.

`: ciao-hook     \ c-addr u -- flag`

This code receives a token that has fallen through the Forth FIND and NUMBER? cycle. If the string does not contain a dot separator it is not a compound statement and false is returned. Otherwise the string is split into a heap allocated token buffer using dot as the delimiter and the following steps taken:

1. If in compile state code is laid to preserve the current THIS.

2. The first token is passed to PROCESS1STTOKEN above to obtain the instance pointer and address provider.

3. NOT IMPLEMENTED YET. The middle tokens should be processed. any of these tokens MUST represent a Class instance or class instance pointer as a member of the current address-provider. Each middle token should compile/execute code to modify THIS and the address provider each time.

4. The Final token is processed according to special rules. It must exist in the name space of the current address provider. If we are defining a method for the current address provider all three scope lists are valid, otherwise the method must be in PUBLIC name space. If successfully located according to scope rules the method entry is passed to METHODTOKENCOMPILE-FROMLIST (see earlier) to compile the invokation code for that method and any applicable operator.

5. The code to restore the saved value of THIS is compiled.

30.24 Instance Creation Primitives

```
: dynamicnew    \ *class -- this
```
The runtime code called for instances created via DNEW. Allocate a block of heap memory large enough for the 2 control cells (*class pointer and vtable pointer) followed by the instance data. The THIS pointer returned is the beginning of the instance data.

```
: staticnew     \ *class "name" -- ; Exec-child: -- this
```
Create a new instance of class called "name" in the dictionary. The instance is a child of CREATE which has a body containing the *CLASS value, a pointer to the instance vtable then the instance data. At runtime a THIS instance pointer is returned, this is 2 cells on from the PFA (IE past the vtable.) Forms the action of NEW when invoked outside of a : definition.

```
: localnew2     \ *class frame-add --
```
I apologise unreservedly for this trick. This definition performs the compile-time tail of localnew. Since LOCALNEW performs a create..does> I cannot add any compile time tail so I tick this definition and push it on the return stack! Sorry ;-)

```
: localnew      \ *class "name" --
```
This definition forms the action of NEW when making a local method. It hacks into VFX locals to create a new entry on the local frame and lays the code necessary (in LOCALNEW2) to setup the two control cells AT RUNTIME. The net result is a very fast and useable named local instance. Implementors beware, this is the most system specified piece of code imaginable.

30.25 Instance Creation

```
: dnew          \ *class -- this
```
State smart definition to create a new class instance on the heap. returns a pointer to the instance which can be stored. A heap allocated instance pointer can be held for as long as required and does not go "out of scope" until explicitly removed with DELETE. Use this type to create a dynamic instance which you can safely return from a method/colon definition. Note that unlike in C++ you must use DNEW in CIAO for heap allocation.

```
: new           \ *class "name" --
```
A state smart definition to create a named instance of a class. When used within a : definition the object is created in a locals frame (one is created if required). When invoked outside of a definition the instance space is ALLOTed from the dictionary. The instance goes out of scope (and is implicitly DELETEd when local) at the same time as the name goes out of scope. For a static instance, i.e. one in the dictionary, it remains in scope for the lifetime of the application (until BYE) whereas for a local instance it is DELETEd and goes out of scope at the end of the definition. Therefore please note that returning a pointer to a local instance *will* break your code - you must DNEW instead. You cannot ever explicitly DELETE a named instance. In future you will be allowed to attempt it and the effect will be to call the destructor method, as will happen when scope is lost anyway.

```
: delete        \ this --
```
DELETE is used to release the memory of a dynamic instance created via DNEW. Memory release for static and local instances is automagic. Any valid destructor is called prior to releasing the memory back to the free-heap. In future performing DELETE on a static or local will simply invoke the destructor.

30.26 AutoVar - An example of a Class

This example shows how to create and use a class called AUTOVAR. This class contains one private data member and two public methods. one method initialises the data store, the other will return the contents of that store and post-increment it.

Defining the class

```
class AutoVar           \ begin a new class definition
private:
        cell:   m_data  \ Where the count will be stored
public:
        meth:   read++  \ The method to read and increment
        meth:   set     \ The method to initialise the count
end-class               \ end definition
```

Coding the Methods

```
: AutoVar::read++       \ -- n ; Method
  1 m_data dup @ -rot +!        \ read and increment data store
;

: AutoVar::set          \ n -- ; Method
  m_data !                      \ write to data store
;
```

Test Code

Here are three test routines which each take an initial value for an AutoVar type and then run the read++ method 10 times writing the result. The first case uses a static instance of AutoVar, the second case uses a local instance, and finally the third case shows how to use a heap allocated instance and how to typecast an object pointer. Note that in CIAO there are separate words for static/local instances with NEW and heap allocation with DNEW.

```
AutoVar new Foo         \ create a static instance of Autovar

: test1                 \ n -- ; Test static instance FOO
  foo.set                       \ init with supplied index
  10 0 do
    cr foo.read++ .             \ read and increment 10 times!
  loop
;

: test2                 \ n -- ; Same thing, local class tho'

  AutoVar new Foobar            \ create local instance of AUTOVAR

  foobar.set
  10 0 do
    cr foobar.read++ .
  loop
;

: test3                 \ n -- ; Third time, heap allocated instance

  AutoVar dnew                  \ create instance and store pointer

  tuck (AutoVar).set            \ use type cast on heap pointer    )
  10 0 do
    cr dup (AutoVar).read++ .                                      )
  loop

  delete                        \ destroy heap instance
;
```

30.27 AutoVar2 - Another Example

AUTOVAR2 is a class derived from AUTOVAR to extend its functionality. AutoVar2 has no publically accessable methods but uses operators to perform the read and initialise.

```
Class AutoVar2 : private Autovar   \ Create new class, inherit from
                                   \ Autovar with all methods in
                                   \ private scope.
     oper: read++                  \ Default operator performs
                                   \ read++ method
  to oper: set                     \ TO operator performs set method
end-class
```

The test procedures could now look like:

```
AutoVar2 new Foo         \ create a static instance

: test1                 \ n -- ; Test static instance FOO
  to Foo                        \ init with supplied index
  10 0 do
    cr foo .                    \ read and increment 10 times!
  loop
;

: test2                 \ n -- ; Same thing, local class tho'

  AutoVar2 new Foobar            \ create local instance

  to Foobar
  10 0 do
    cr foobar .
  loop
;
```

30.28 Class Library

The following code documents the beginning of an MFC style class library for CIAO.

30.28.1 Base Operators

The following operators have been defined and are used through out the base classes whenever applicable. Each class will document its use of these operators.

`operator: ++                    \ --`
Increment by 1

`operator: --                    \ --`
Decrement by 1

`operator: cout<<                \ --`
Display applicable output

`operator: cout<<hex             \ --`
Display applicable output in hex

`operator: xywh->                \ x y width height --`
Store X Y WIDTH HEIGHT parameters.

`operator: lprect->              \ *Rect --`
Store X Y WIDTH HEIGHT obtained from a RECT structure.

`operator: []                    \ index -- char`
Get array element at index.

`operator: []to                  \ index val --`
Set element at index: <idx> <val> []to <class>

`operator: (LPCTSTR)             \ -- z$`
Get contents as a zero-termintated string.

`operator: (LPCTSTR)to           \ z$ --`
Set contents from a 0 terminated string pointer.

```
operator: +=                    \ instance-pointer --
```
Concatenate/Add from a class of same type.

```
operator: (LPCTSTR)+=           \ z$ --
```
Concatenate from a 0 terminated string.

30.28.2 Primitive Types

This collection of classes represents the primitive data types found in C++. They can be though of as extended Forth VALUEs.

INT

This data type represents a simple number which is basically equivalent to a CELL. It has no methods publically callable but simply uses operators to access.

The following operators have been assigned to methods for this class.

`<default>`
: No operator, returns contents.

`to` Set content from stack item.

`addr` Get the address rather than contents.

`++` Increment by 1.

`--` Decrement by 1.

`cout<<` Write contents to console.

`cout<<hex`
: Write contents as hex to console.

30.28.3 Windows Types

Defines some simple classes for Windows data-types. Most simple types under Windows are just 32 bit numbers. Therefore the following types are all simply private scope derived from the INT type documented before.

```
LPVOID             HANDLE          HWND            HMENU
HINSTANCE          LPCTSTR         LONG            DWORD
```

30.28.4 Windows Structures

The following structures are defined using standard Windows names. all data members are one of the previously defined Windows types.

```
RECT       CREATESTRUCT            POINT
```

30.28.5 CPOINT - Point Class

This is simply a class version of the POINT structure. The reason for the separation is that the STRUCT{ version can be typecast from an OS supplied point-struct into a CIAO POINT struct or CPOINT class

The CPOINT class is publicly derived from POINT with a method and operator for TO supplied which takes another point as the source.

30.28.6 CRECT - Rect Class

This is a class version of the RECT structure. The CRECT class is publicly derived from RECT with methods and operators.

```
class CRect : public Rect
public:
                      meth:   Width
                      meth:   Height
                      meth:   SetXYWH
                      meth:   SetLPRECT
                      meth:   dump
   ->                 oper:   SetLPRect
   xywh->             oper:   SetXYWH
   lprect->           oper:   SetLPRECT
   cout<<             oper:   dump
```

30.28.7 CString - Dynamic String Class

A CString object contains a variable-length sequence of characters. It also provides functions and operators which allow for easy to Concatenation and comparison operators, together with automatic memory management. CString objects are far easier to use than ordinary character arrays.

CString is based on the FORTH char data type.

CString Objects have the following useful characteristics:

- CString objects can grow as a result of concatenation operations. The memory management is automatic.
- You can easily substitute CString objects for const char* and LPCTSTR function arguments by using the (LPCTSTR) operator or its) associated method GetLPCTSTR.
- You can lock a CString object to provide a character buffer at a fixed address and size for hacking.

Variable Type Arguments

Some of the members of this class can take either a character or a zero-terminated string as a parameter. This is autodetected by the simple assumption that any value greater than the maximum value storable in a char is an array pointer. For 8 bit character systems this means a pointer cannot be in the range 0..255.

The CString Class Members

`: CString::ResizeBuffer \ rsize --`
Resize the current string buffer memory to RSIZE chars.

`: CString::Empty        \ --`
Release all string memory and return to init state.

`: CString::GetAt        \ idx -- char`
Return the ascii character at IDX position in the string.

`: CString::GetLength    \ -- n`
Return the length of the current string.

`: CString::GetLPCTSTR   \ -- z$`
Return a pointer to the string as a zero-terminated. After obtaining this pointer, any operation which modifies the string may destroy this pointer.

`: CString::IsEmpty      \ -- BOOL`
Return TRUE if there is no string information.

`: CString::SetAt        \ idx char --`
Place character CHAR at the IDX position in the string.

`: CString::Add          \ *CString --`
Add the contents of the CString class pointed to onto the end of the current string.

`: CString::AddLPCTSTR   \ z$ --`
Add the supplied zero terminated string to the end of the current.

`: CString::SetLPCTSTR   \ z$ --`
Replace the current string with the supplied zero terminated one.

`: CString::to           \ *CString --`
Replace the current string with the contents of the CString class whose pointer is supplied.

`: CString::Compare      \ z$ -- flag`
Compare the current string with the zero-terminated string supplied.

`: CString::CompareNoCase \ z$ -- flag`
As CString::Compare except character case is ignored.

`: CString::Mid          \ first count -- *CString(dynamic)`
Return a new dynamic instance pointer for a CString which contains a substring of the current. COUNT characters from index FIRST are copied.

`: CString::Left         \ count -- *CString(dynamic)`
Return a new dynamic instance pointer for a CString which contains a substring of the current. COUNT characters are copied from the start (left) of the string.

`: CString::Right        \ count -- *CString(dynamic)`
Return a new dynamic instance pointer for a CString which contains a substring of the current. COUNT characters are copied from the end (right) of the string.

`: CString::Delete       \ index count -- newlen`
Remove COUNT characters from the string starting at the INDEX position. If count+index exceeds the string length it is trucated. Also returns the new length of the string after the delete.

`: CString::Insert       \ index z$ -- int | index char -- int`
Passed either an index and a z$ or an index and a character this will perform an insert operation. The string or character supplied is inserted starting at the original offset INDEX. Returns the new length of the string.

`: CString::MakeUpper    \ --`
Convert the classes string data to upper case where possible.

`: CString::MakeLower    \ --`
Convert the classes string data to lower case where possible.

Operator Associations

The following operators have been assigned to methods for this class.

`[]` GetAt – Get character at specified index.

`[]to` SetAt – Set chatacter at specified index.

`(LPCTSTR)`
GetLPCTSTR – Return 0terminated string pointer.)

`to` to – Assign from another CString.

`(LPCTSTR)to`
SetLPCTSTR – Assign from a 0 terminated string.)

`+=` Add – Append from another CString.

`(LPCTSTR)+=`
AddLPCTSTR – Append from a 0 terminated string.)

31 Internationalisation

Internationalisation often requires support for strings longer than the 255 characters supported by counted strings in the 8 bit character set used by VFX Forth during application development. Such strings may also not be in the character set or size used by the application developer.

Internationalisation often requires third parties to be able to convert text strings without having to recompile the application.

Forth system developers and vendors need to make their systems compatible with their clients existing approaches to internationalisation.

This implementation supports all these requirements, and is a compatible superset of the current ANS Forth Internationalisation proposals, which are available from the downloads section of the MPE web site at: http://www.mpeforth.com

If you are using this software with MPE's VFX Forth system, the source code is in the file LIB\INTERNATIONAL.FTH.

MPE acknowledges the help and support of Construction Computer Software, Cape Town, South Africa, in the design of this software. The CCS application has been internationalised for many years, and their experience has been invaluable, both in defining the draft ANS standard and in developing this code.

31.1 Long string parsing support

```
: parse/l       \ char -- c-addr len ; like PARSE over lines
```

Parse the next token from the terminal input buffer using <char> as the delimiter. The text up to the delimiter is returned as a c-addr u string. PARSE/L does not skip leading delimiters. In order to support long strings, PARSE/L can operate over multiple lines of input and line terminators are not included in the text. The string returned by PARSE/L remains in a single global buffer until the next invocation of PARSE/L. PARSE/L is designed for use at compile time and is not thread-safe or winproc-safe.

31.2 Data structures

31.2.1 Rationale

Although internationalised strings may be referenced by the addresses of suitable data structures, these addresses will change from build to build of the application. The implementation here permits strings to be given a number which does not change between builds. Together with a compile-time hook which can generate a text file in the development language, application strings can be translated in external text files without rebuilding the application. This is required in situations in which translation is performed locally by dealers or by users themselves.

The /TEXTDEF structure described below permits messages to be accessed either by message number or by the address of the structure.

31.2.2 /TEXTDEF structure

Internationalisation of messages relies on a data structure /TEXTDEF. The /TEXTDEF structure contains a link to the previous TEXTDEF or #TEXTDEF definition, a message identifier which is 0 for non-databased strings in the ISO Latin1 coding, the address of the text, and the length of the text in bytes. The text is followed by two zero bytes, and the text is long aligned. The /TEXTDEF structure is a superset of the /ERRDEF structure used for error messages by VFX Forth.

The words #TEXTDEF and ERR$ are DEFERred. #TEXTDEF is used by TEXTDEF. The user can install alternative versions of these words for internationalised applications. In this context, #TEXTDEF and friends can be used as the basis of any text handler that requires translation. Note that #TEXTDEF can be modified so that a message file is produced at compile time, and ERR$ modified so that the message file is accessed at run time. Similarly, providing that the application language is correctly handled, the run time can access translated messages in other languages, character sets and character sizes.

The messages are linked into the same chain as is used for all error strings that can be internationalised. This chain is anchored by the variable TEXTCHAIN.

```
struct /textdef \ -- len ; DOES NOT include constant definition
  int td.value          \ value that identifies string
  int td.link           \ link to previous TEXTDEF td.link field
  int td.id             \ 0 or message ID
  int td.caddr          \ address of text string
  int td.len            \ length of text string
  int td.lenInline              \ length of inline text string in bytes
end-struct
```

31.2.3 String structure

31.3 Creating and referencing LOCALE strings

In this implementation, the ANS locale string identifier "lsid" is a pointer to a /TEXTDEF structure.

`defer l$CompileHook     \ ^textdef --`

A DEFERred hook that the user can modify to produce additional data at compile time. For example, the hook is commonly replaced by code that generates a text file in the development language. This text file then serves as the basis for translation to other languages.

`: L$",          \ n -- ; compile a long string`

This can be thought of as a multiline version of ",. First a /TEXTDEF structure is created. Then it collects multiline text and lays down an inline string with two zero bytes as termination. The start of the string is aligned on a four-byte boundary. The end of the string is padded to a four-byte boundary.

`defer #TEXTDef  \ n -- ; -- n`

Define a constant and associated message in the form: <n> #TEXTDEF <name> "<text>". Execution of <name> returns <n>.

`: NextText      \ -- addr`

Returns the address of the variable holding the next constant used to identify an internationalised string.

`: NextText#     \ -- n`
Return the contents of NEXTTEXT and increment NEXTTEXT.

`: TextDef       \ -- ; -- n ; used as throw/error codes`
Define a constant and associated message in the form: TEXTDEF <name> "<text>". Execution of <name> returns the constant automatically allocated by NEXTTEXT#.

`: l$find        \ n -- struct|0 ; produce pointer to TEXTDEF structure`
Given a message number n, return the address of the /TEXTDEF structure containing its details.

`: l$count       \ lsid -- c-addr u`
Given a /TEXTDEF structure, the address and length in bytes of the text string are returned.

`: l$addr        \ lsid -- c-addr`
Given a /TEXTDEF structure, the address of the text string is returned.

`: (l$")         \ -- lsid`
The runtime action of L$" to return the address of the /TEXTDEF structure associated with the string compiled by L$".

`: L$"           \ -- ; -- lsid`
Used inside a colon definition to compile a string that will be internationalised. At run time the address of the TEXTDEF structure will be returned.

`: LS"           \ -- ; -- caddr u`
Used to compile or extract a long string. When used during compilation L$", is used to a lay down a string for internationalisation. At run time the address and length of the string are returned.

`: ZLS"          \ -- ; -- c-addr`
Used to compile or extract a zero terminated long string. When used during compilation L$", is used a lay a string for internationalisation. At run time the address of the string is returned.

31.4 ANS LOCALE word set

In this implementation, the ANS locale string identifier "lsid" is a pointer to a /TEXTDEF structure.

`defer set-language      \ lang -- ior`
Set the current language code. At the very least, the action of this word must be to set the variable <LANGUAGE>. The action may also include updating the string data in the TD.CADDR and TD.LEN fields of all the /TEXTDEF and /ERRDEF structures. If the operation succeeds, the returned ior is 0. If the operation fails, the returned ior is non-zero and the meaning of the ior is implementation dependent.

`: get-language  \ -- lang`
Return the current language code.

`defer set-country       \ country -- ior`
Set the current country code. At the very least, the action of this word must be to set the variable <COUNTRY>. The action may also include updating locale-sensitive routines such as date and time display formatting words. If the operation succeeds, the returned ior is 0. If the operation fails, the returned ior is non-zero and the meaning of the ior is implementation dependent.

`: get-country   \ -- country`

Return the current country code.

`: l"            \ -- ; -- lsid ; L" <native text>"`

A locale-sensitive version of C" which returns an lsid (string indentifier) at run-time. The native text may be compiled inline
Interpetation: The interpretation semantics for this word are undefined.

Compilation: \ "ccc<quote>" – Parse ccc delimited by a " (double-quote) and append the run-time semantics given below to the current definition.

Runtime: \ – lsid Return lsid, an identifier for a locale string. Other words use lsid to extract language specific information.

`: LOCALE@       \ lsid -- addr len(au)`

Return the address and length in address units of the string (in the current language) that corresponds to the native string identified by lsid. The format of the string at addr is implementation dependent. The length of the string is returned in address units so that it may be copied by `MOVE` without knowledge of the character set width.

`defer SUBSTITUTE       \ i*x addr1 len1 addr2 len2 - j*y addr2 len3`

Perform macro substitution on the lstring at addr1/len1 placing the result at lstring addr2/len2, returning addr2 and len3, the length of the resulting string. Ambiguous conditions occur if the resulting string will not fit into addr2/len2, or macro text cannot be found, or if the lstring at addr2/len2 overlaps the lstring at addr1/len1. Macros may take parameters from the Forth data stack.
When a macro name delimited by escape characters (see SET-ESCAPE) is encountered by SUBSTITUTE, the following action occurs:

1) If the name is a valid macro name, a locale and implementation dependent action occurs

2) If the name is null, a single escape character is substituted

3) In all other cases an ambiguous condition exists.

`: (substitute)  \ srce slen dest dlen -- dest dlen'`

Expand source using macros. Note that this version is simplistic, performs no error checking, and requires a global buffer.

`defer set-macro \ addr len c-addr u --`

Define the localised string addr/len in address units as the text to substitute for the macro of the name (in the development character set) c-addr/u. If the macro does not exist it is created.

`: SET-ESCAPE    \ locale-char --`

Set the macro escape character to be the localised character locale-char. By default it is the ASCII % character if it is available in the application character set.

`: GET-ESCAPE    \ -- locale-char`

Return the macro escape character locale-char. By default it is the ASCII % character if it is available in the application character set. See SET-ESCAPE.

31.5 ANS LOCALE extension word set

In this implementation, the ANS locale string identifier "lsid" is a pointer to a /TEXTDEF structure.

```
defer LOCALE-INDEX      \ lsid --
```
Updates the internal data structure. Useful if structures are added and changes to internal structures are required.

```
: LOCALE-LINK   \ lsid1 -- lsid2
```
Given the address of one LOCALE structure, returns the address of the next.

```
defer LOCALE-TYPE       \ addr len --
```
Displays the LOCALE string whose address and length in address units are given.

```
: NATIVE@       \ lsid -- c-addr len
```
Given a LOCALE structure, returns the address and length of the corresponding DCS native string that was compiled by L".

31.6 Windows language support

Windows contains a large number of predefined language constants of the form LANG_xxx and SUBLANG_xxx. A Windows locale is identified by merging a pair of these as described below.

```
+-------------------------+-------------------------+
|     SubLanguage ID      |   Primary Language ID   |
+-------------------------+-------------------------+
15                    10  9                         0   bit
```

These constants can be viewed from VFXForth by using:

```
SIM LANG_
SIM SUBLANG_
```

These codes use 0 as the current or neutral code, which matches using 0 as the language code for the development character set, which is ISO Latin 1 for VFX Forth. In this set, the seven bit ASCII character set defined by ANS Forth represents characters 0..127.

```
: langID        \ primary secondary -- langid
```
Generate a Windows language code from the primary and secondary codes, e.g.

```
LANG_SPANISH SUBLANG_SPANISH_MEXICAN langid
```

32 Obsolete words

The following words are now obsolete and have been removed from the VFX kernel. If their use is required, they may be found in *LIB\OBSOLETE.FTH*.

`: ALIGN&ERASE   \ --                                    MPE.0000`

Align the dictionary pointer, zeroing any intermediate memory. This word is now obsolete as `ALIGN` now performs the same action.

`: HALF-ALIGN&ERASE      \ --`

`HALF-ALIGN` the dictionary pointer, zeroing any intermediate memory. This word is now obsolete as `HALF-ALIGN` now performs the same action.

`: M/MOD          \ d1 n2 -- rem quot                      MPE.0000`

Signed version of `UM/MOD`. This word is obsolescent and should be replaced by `FM/MOD` (floored division) or `SM/REM` (symmetric division).

`: CONVERT        \ ud1 c-addr1 u1 -- ud2 c-addr2         6.2.0970`

An obsolescent word corresponding to `>NUMBER DROP`.

`cell +USER SPAN \ -- addr`

Required by `EXPECT` and Forth 83 systems.

`: EXPECT         \ c-addr +n --                           6.2.1390`

Wait for input from the console. Data is stored in the buffer at *c-addr* for upto *n* characters. After input is complete due to either a full buffer or a carriage return, the length of the input string is also stored in the variable `SPAN`. This word is marked as obsolescent in the ANS specification, and new code should use `ACCEPT` instead.

`: v-find         \ caddr voc-xt -- cfa/cfa/caddr +1/-1/0      SYS.0000`

A near equivalent to `SEARCH-WORDLIST` retained in VFX Forth as a concession to PFW 2.x users.

`: all            \ -- -1 -1`

A ProForth 2.x compatibility word.

`: from-file      \ start end "<name>" --`

A ProForth 2.x compatibility word.

`: pto            \ --                                     MPE.0000`

Skip parsing until an ASCII 12 pagethrow is encountered.

`: winapphandle@ \ -- hwnd`

Return the console parent frame window handle.

`: OFFSET         \ x "<spaces>name" -- ; Exec: n -- n+x*4        MPE.0000`

Create a new offset called *name*. On execution of *name* the supplied address will be incremented by *x* cells.

`: BOFFSET        \ x "<spaces>name" -- ; Exec: n -- n+x          MPE.0000`

As with `OFFSET` except the increment is specified in bytes rather than cells.

`: instance       \ n -- ; -- addr                         MPE.0000`

Create a named instance of a named structure. A memory buffer *n* bytes long called *name* is built. When *name* is executed the address of the buffer is returned. Use `buffer:` instead.

32.1 Removed from VFX Forth v4.0

`: <<n \ x1 u -- x2` MPE.0000

Logically shift *X1* by *U* bits left. Use `lshift` instead.

`: >>n \ x1 u -- x2` MPE.0000

Logically shift *X1* by *U* bits right. The result of shifting by more than 31 bits is undefined. Use `rshift` instead.

`: s>>n \ x1 u -- x2` MPE.0000

Shift *x1* right by *u* bits, filling with the previous top bit. An arithmetic right shift. The result of shifting by more than 31 bits is undefined. Use `arshift` instead.

`: ($+) \ c-addr u $dest --` MPE.0000

Add the string described by *C-ADDR/U* to the counted string at *$DEST*. The strings must not overlap. Use `APPEND` instead.

`: z>here \ --`

Lay the counted string at `PAD` into the dictionary at `HERE`.

33 Migrating to VFX Forth

VFX Forth is a major technical upgrade from the ProForth 2.x and other threaded-code Forth compilers. This section describes some of the "gotcha's" in porting code to VFX Forth from ProForth, pre-ANS systems and non-optimising compilers.

VFX Forth is brutally intolerant of programming errors. We have found that this approach, sometimes described as "crash early and crash often", leads to code with fewer lurking bugs. One customer who converted a large application to VFX Forth found that VFX Forth crashes revealed bugs that had been lurking for many years.

33.1 VFX generates native code

VFX Forth uses native code compilation with aggressive optimisation. This is perhaps the single biggest difference between VFX Forth and ProForth. Execution speed is a primary goal, and the benchmark figures show that we have achieved it.

Extra care should be exercised with any source code which requires knowledge of the underlying architecure. This will particularly impact definitions which cause compilation and assembler fragments. Many words are provided in VFX Forth to hide the implementation details.

33.2 VFX uses absolute addresses

The VFX Kernel runs in absolute address space just like any other Windows/Mac/Linux/DOS application. There is no need to convert Windows addresses to Forth ones using words such as `REL>ABS` and `ABS>REL` found in ProForth 2.x and some derivatives.

33.3 VFX is an ANS standard Forth

The VFX Kernel is based on the ANS language specification rather than Forth83. This introduces a number of minor differences in the behaviour of the system and the code produced.

33.4 COMPILE is now IMMEDIATE

Previous MPE implementations used a non-immediate version of `COMPILE` which has "unpicked" the following CALL instruction at run-time. This behaviour has been changed.

33.5 Comma does not compile

VFX Forth is a native code compiler. Threaded code systems allowed compilation by "comma-ing" a CFA into the dictionary. This is no longer a valid method of generating code. The ANS word `COMPILE,` should be used instead. Also the system must be in "compile state" when `COMPILE,` is used.

33.6 COLON and CURRENT

Under VFX Forth, the `CURRENT` definitions wordlist is **not** modified by COLON (`:`). Also note that `:` is no longer immediate.

33.7 The Assembler is built-in

The assembler within VFX is built in as part of the kernel since is used by the code generator. The `ASM` and `UNHOOK-ASM` directives found in ProForth are redundant. VFX Forth does implement these two directives in a compatibility layer which will write a warning message to the console hinting at non-portable code.

33.8 The Inner Interpreter is different

ProForth 2.x applications which relied on or modified the behaviour of the interpreter will **not** port. VFX Forth has a unified interpreter rather than the `C-LOOP` and `I-LOOP` pair.

Both `QUIT` and `INTERPRET` are defered for those applications which must override the interpreter/compiler behaviour of the system but they should not be used for any new code.

33.9 The FROM-FILE word has gone

ANS specifies the ability to include source-code from ASCII text files. This behaviour is implemented in VFX Forth. The MPE `FROM-FILE` handler code in ProForth 2 is not supported. Programmers should look at the ANS definitions `INCLUDED`, `INCLUDE-FILE` and `REFILL` to understand the new approach.

33.10 Generic I/O

Although `KEY` and `EMIT` and friends are still `DEFER`red, we **strongly** recommend that you leave them alone because many system tools rely on Generic I/O. This means that you should build Generic I/O tables for all devices you want to use. You can use the examples in *LIB\GENIO* as models.

33.11 External API Linkage

The method used for linking to external library calls has changed radically. `LIBFUNCTION:` is still supported in the compatibility layer but all new imports should use the `EXTERN:` syntax described in this manual. The new syntax allows you to bind both C and PASCAL convention definitions, as well as making the job of converting C header files easier.

33.12 DLL generation

The mechanism used by VFX Forth is completely different.

33.13 Windows Resource Descriptions

All the original MPE syntax for defining Windows resources such as menus and dialog-boxes has been removed in favour of the new parser to handle Windows RC files.

The conversion of "old-style" resources to the new ones must be done "by-hand". The new syntax is cleaner and easier to maintain as well as being largely compatible with 3rd party resource editors.

33.14 ANS Error Handling

Error handling in VFX is done using the ANS `CATCH` and `THROW` mechanism. The words `ERROR` and `?ERROR` from ProForth 2.x are gone. Please read the section on exception handling both in this manual and the ANS Forth Standard.

33.15 Obsolete words

The file *LIB\OBSOLETE.FTH* contains definitions for many ProForth 2.x words which are not present in VFX Forth.

34 Rebuilding VFX Forth for Linux

Users of the VFX Forth Professional and Mission editions have all the source code and tools needed to rebuild the system. VFX Forth Mission includes the source code for the tools as well as the Forth source code.

The source code is found in the *Sources* directory. Tools are in the *Tools* directory. The build process is controlled by a set of Linux shell scripts in the *Sources* directory itself. Some of these scripts may contain hard-coded paths and should be checked and edited before running the scripts. The resulting executables will be placed in the directory *Sources\Images*.

34.1 Rebuilding VFX Forth

The full build is performed under Linux by executing

```
./RebuildLin.sh
```

from the *Sources* directory.

The build is performed in three stages:

- First stage - rebuild the kernel using the i386+ Forth Cross Compiler
- Second stage - use the kernel to generate the base console
- Third stage - **Future version**: use the base console to generate the GUI development environment.

Short cuts are available through batch files described later.

34.1.1 Kernel

The core kernel is cross-compiled from the *Sources/Kernel* folder by executing

```
./mlinux.sh
```

which runs the cross compiler twice to produce the Pentium and 386 kernels *vfxkern.elf* and *vfxkern386.elf*.

34.1.2 Second stage

The second part of the build produces the base version of VFX Forth. This part of the build occurs in *Sources/VFXBase*. Run:

```
./mlinux.sh
```

It compiles the second stage builds to produce *vfxlin* and *vfxlin386* in the the *Sources/Images* directory. This process can also be performed from the *Sources* directory:

```
./Stage2Lin.sh
```

34.1.3 Third stage

To be defined

34.2 Manuals

The PDF manuals require a LaTeX with TexInfo distribution on a Linux box.

The HTML and PDF manuals are produced by DocGen. Change to the *Sources\Manual* directory and run

```
./MakeLinux.sh
```

34.3 Rebuilding the tools

The tools source code supplied with the Mission edition will be found in the *TOOLS\SRC* directory. Each subdirectory will contain a make file *MAKEFILE* or a shell script *MAKExxx.sh*. Tools written in C will have been compiled by gcc.

34.3.1 Rebuilding the libraries

The two shared libraries, *libmpeparser.so.0* for INI files and *vfxsupp.so.1.0.1* for Linux constants, very rarely need rebuilding.

INI file library

The INI file support library enables persistent storage of configuration data using Windows-style INI files. The code accesses a derivative of the iniParser v3.0b shared library published by Nicholas Devillard at *http://ndevilla.free.fr/iniparser/*, where the latest version may be found. Note that the MPE versions differ from this version, but are upward compatible. We have submitted our changes to the author.

The source code for the MPE versions (Linux and Windows) are in *Sources/Tools/iniparser3.0b*. For the Linux version, use *Makefile* in the root of the directory. For the Windows version, switch to the *src.win* folder and run *make.bat* which is for use with VC6.

Linux constants library

The *vfxsupp* library allows the Forth system to look up Linux constants from header files without applications having to be bloated by a vast number of headers. The number handling mechanism in VFX Forth searches the library if the text cannot converted as a number.

The sources are in *Tools/VfxSupport/Lin32*. The library is built using *Makefile*. If you get an error, especially after adding a new header file to *headerlist.h*, it is nearly always because the constant cannot be correctly evaluated by the Forth application, e.g. because it is a pointer to a text string. In this case, add the item to *headerlist.exclude* and rebuild. Keep going until there are no more errors.

The essence of the process is that the header files are run through the C pre-processor to produce *headerlist.i.orig*, filtered by an awk script to produce *headerlist.i*, and then incorporated into the shared library.

34.3.2 Packaging

The packaging script was written by Kelly Dunlop and then modified by Vic Watson and Stephen Pelc. Any faults in this script are entirely the responsibility of MPE. Without the support and persistence of Kelly and Vic, the VFX Forth for Linux packages would not exist. If you have the choice, leave maintenance of the packaging scripts to other people.

Compared to packaging Windows programs, packaging Linux applications is a nightmare. Packaging 32 bit applications for general release requires five different releases, depending on the default installer and CPU:

- RPM32 - for installation on 32 bit RedHat Linuces and derivatives.
- RPM64 - for installation on 64 bit RedHat Linuces and derivatives.
- DEB32 - for installation on 32 bit Debian Linuces and derivatives.
- DEB64 - for installation on 64 bit Debian Linuces and derivatives.
- Tarball - for Linuces without RPM or DEB installer support, and for those that have peculiar requirements for directory structure.

As a result of having three editions of VFX Forth, as of November 2013, only tarballs are supported.

Everything is done by *PackageLin32.sh*. The script is heavily commented. If you change it, on your own head be it. The RPM builds require the file *.rpmmacros* in your home directory. You will find a sample *rpmmacros.txt* in the *Scripts* directory with default sets of the scripts. See also the directory *Scripts/extras*.

34.4 Mission edition builds

The Mission edition of VFX Forth contains source code and tools for everything to do with VFX Forth, including the full build, release and packaging scripts used by MPE to issue the software. Assuming that the Mission release is in directory called *VfxForth.dev*, the complete process is performed by changing to that directory and running:

```
./MakeLin32.sh
```

which runs *Scripts.Lin32/FullLin32.sh* that calls many other scripts, including:

- *getXC386.sh* - copy the cross compiler into the *Tools* directory.
- *getGenDocs.sh* - copy documentation shared with other MPE products.
- *Manual/MakeLinux.sh* - builds the Linux HTL and PDF manuals and helper files.
- *RebuildLin.sh* - rebuilds the executable files.
- *ReleaseLin.sh* - builds the release folders from which the packages are built.
- *PackageLin.sh* - builds the Linux packages and tarball.
- *DistribLin.sh* - creates/updates a repository on a server and copies the packages to it.

35 Further information

35.1 MPE courses

MicroProcessor Engineering runs the following standard courses, which can be held at MPE or at your own site:

- **Architectual Introduction to Forth** (AIF): A three-day course for those with little or no experience of Forth, but with some programming experience. The AIF course provides an introduction to the architecture of a Forth system. It shows, by teaching and by practical example how software can be coded, tested and debugged quickly and efficiently, using Forth's interactive abilities.
- **Embedded Software for Hardware Engineers** (ESHE): A three-day course for hardware and firmware engineers needing to construct real-time embedded applications using Forth cross-compilers. Includes multitasking and writing interrupt handlers.

Custom courses are available

- **Quick Start Course** (QSC): A very hands-on tailored course on your site using your own hardware, and includes installation of a target Forth on your hardware, approaches to writing device drivers, designing a framework for your application and whatever else you need. The course is usually three days long.
- Other custom courses we provide are for Open Boot and Open Firmware. These are derived from the AIF course above.

35.2 MPE consultancy

MPE is available for consultancy covering all aspects of Forth and real-time software and hardware development. Apart from our Forth experience, MPE staff have considerable knowledge of embedded hardware design, Windows, Linux and DOS.

Our software orbits the earth, will land on comets, runs construction companies, laundries, vending machines, payment terminals, access control systems, theatre and concert rigging, anaesthetic ventilators, art installations, trains, newspaper presses and bomb disposal machines.

We have done projects ranging from a few days to major international projects covering several years, continents and many countries. We can operate to fixed price and fixed term contracts. Projects by MPE cover topics such as:

- Custom compiler developments, including language extensions such as SNMP, and new CPU implementations,
- Custom hardware design and compiler installations,
- Portable binary system for smart card payment systems,
- Machinery controllers,
- Connecting instrumentation to web sites,
- Virtual memory systems,
- Code porting to new hardware or operating systems.

We also have a range of outside consultants covering but not limited to:

- Communications protocols

- Windows device drivers
- All aspects of Linux
- Safety critical systems
- Project management (including international)

35.3 Recommended reading

A current list of books on Forth may be found at:

`http://www.mpeforth.com/books.htm`

For an introduction to Forth, and all available in PDF or HTML:

- "Programming Forth" by Stephen Pelc. About modern Forth systems.
- "Starting Forth" by Leo Brodie. A classic, but very dated.
- "Thinking Forth" by Leo Brodie. A classic.

For more experienced Forth programmers:

- "Object Oriented Forth" by Dick Pountain
- "Scientific Forth" by Julian Noble

Other miscellaneous Forth books:

- "Forth Applications in Engineering and Industry" by John Matthews
- "Stack Machines: The New Wave" by Philip J Koopman Jr

All of these books can be supplied by MPE.

Index

/

:

;

<

=

>

?

@

[

]

^

_

B

C

D

E

F

G

H

I

M

N

O

P

Q

R

S

T

X

Z