Using Win32Forth

By Bill Ragsdale

Contents

[A Bit of Forth History 3](#_Toc54000562)

[Introduction 3](#_Toc54000563)

[On Standardization 4](#_Toc54000564)

[On Improvements 4](#_Toc54000565)

[How To Install 4](#_Toc54000566)

[Your first Forth program 5](#_Toc54000567)

[Basic Resources 5](#_Toc54000568)

[Interacting With The Outer (Text) Interpreter 6](#_Toc54000569)

[Inner Interpreter 6](#_Toc54000570)

[Some Words To Try 6](#_Toc54000571)

[Documentation 7](#_Toc54000572)

[Getting Familiar 7](#_Toc54000573)

[The Stack 9](#_Toc54000574)

[Words and Stack Notation 9](#_Toc54000575)

[Stack Manipulation 10](#_Toc54000576)

[64 bit Stack Manipulation 11](#_Toc54000577)

[Floating Point Stack Operations 11](#_Toc54000578)

[Return Stack Operations 11](#_Toc54000579)

[Input Formats and Number Bases 12](#_Toc54000580)

[Numeric Formats 12](#_Toc54000581)

[Floating Point Input And Output (x86 format) 13](#_Toc54000582)

[Display and Printing Formats 14](#_Toc54000583)

[Pictured Numeric Output 15](#_Toc54000584)

[Case Sensitivity 15](#_Toc54000585)

[More on Forth Words 16](#_Toc54000586)

[Tool and Utility Words 16](#_Toc54000587)

[Memory Addressing 17](#_Toc54000588)

[Forth’s Addressing Conventions 17](#_Toc54000589)

[Memory operators 18](#_Toc54000590)

[Data Structures 18](#_Toc54000591)

[Constants 19](#_Toc54000592)

[Variables 19](#_Toc54000593)

[2VAR and FVAR? 20](#_Toc54000594)

[VALUE Variable 20](#_Toc54000595)

[Data In Memory 20](#_Toc54000596)

[Examples of Data Storage In Memory 20](#_Toc54000597)

[Word Types 22](#_Toc54000598)

[Text Strings 23](#_Toc54000599)

[String Utility Words 23](#_Toc54000600)

[About File Name 26](#_Toc54000601)

[The Editor 26](#_Toc54000602)

[Keyboard Shortcuts 26](#_Toc54000603)

[User Support words 28](#_Toc54000604)

[Comments 29](#_Toc54000605)

[Cell Size and Manipulation 29](#_Toc54000606)

[Reading and writing memory 30](#_Toc54000607)

[Locating Words In The Dictionary 30](#_Toc54000608)

[Vocabularies 32](#_Toc54000609)

[Word Searches 32](#_Toc54000610)

[Forgetting Words In The Dictionary 33](#_Toc54000611)

[Dictionary Layout and Access 34](#_Toc54000612)

[Advanced Topics 34](#_Toc54000613)

[Header structure 34](#_Toc54000614)

[Structure of Dictionary Fields (as of 2003) 35](#_Toc54000615)

[Use Of Header Fields 36](#_Toc54000616)

[CREATE and DOES> 37](#_Toc54000617)

[DEFERed words 38](#_Toc54000618)

[Use of ;CODE 38](#_Toc54000619)

[CODE Words 38](#_Toc54000620)

[Recursion 38](#_Toc54000621)

[ontrol Flow, Decisions 39](#_Toc54000622)

[Execution Chains 42](#_Toc54000623)

[CATCH & THROW 42](#_Toc54000624)

[Manipulation of memory 43](#_Toc54000625)

[Interpretation 43](#_Toc54000626)

[DOS Commands 44](#_Toc54000627)

[Conditional Execution and Compilation 44](#_Toc54000628)

[Error Conditions 45](#_Toc54000629)

[Manipulating the Top of Stack 46](#_Toc54000630)

[User Area 46](#_Toc54000631)

[Using Floating Point 47](#_Toc54000632)

[Float-point Representations 47](#_Toc54000633)

[Summary 47](#_Toc54000634)

[Floating Point Internals and Details 48](#_Toc54000635)

[File Access Details 48](#_Toc54000636)

[Assembly Code Level Internal Details 49](#_Toc54000637)

[Bibliography 51](#_Toc54000638)

[On-line Resources 51](#_Toc54000639)

[About The Author 51](#_Toc54000640)

[Open Items and Questions 52](#_Toc54000641)

[Note on setting fonts 52](#_Toc54000642)

\*\*\*\*\* THIS IS A WORK IN PROCESS \*\*\*\*\*

\*\*\*\*\*\*\* EXPECT IRREGULARITIES \*\*\*\*\*\*

## A Bit of Forth History

In 1978, I (Bill Ragsdale) and several local personal-computer (a term developed much later) enthusiasts attended a demonstration of Forth by Forth, Inc. We marveled at the claims made of development speed, compact structure, execution speed and interactivity. One of our group verified the statements as he had used microForth on a Hughes 1802 computer. Forth, Inc. stated they were not going to pursue the personal computer (hobbyist) market.

As a result, I teamed with Major Robert Selzer and developed a prototype Forth for my business, Dorado Systems based in the design and structures of Forth, Inc’s microFORTH.. This was later formalized into fig-FORTH and placed in the public domain. The Forth Interest Group sponsored a series of Implementation Workshops in which I guided the development of about eight fig-FORTH implantations on processors such as 8080, 6800, 6502, PDP-11, Texts Instruments TMS9900, Data General, Computer Automation, etc. This work evolved into MVP Forth (following the Forth-79 Standard), the Laxon Perry’s Forth-83 (following the 1983 Forth Standard) and finally Win32FORTH (following the ANSI Forth Standard). Andrew McKewan wrote the Forth Kernel, MetaCompiler and more (?) and passed the project to Tim Zimmer. Later Andrew McKewan continued the development. As a side note Andrew began his Forth career with me at Dorado Systems, continuing ahead with his own ground-breaking work. The work has been carried to this day by a group of dedicated programmers.

## Introduction

ALERT = = If you are viewing this using Microsoft Word, then select View, Views, Outline. Then Outline Tools, Show Level: Level 2. You will see an overview by section for easy location of content.

Win32Forth original distribution from 1995 labeled ‘Zimmer’s last V 4.2.0516’ is stable and unchanging. It will run on XP through to Windows 10 (8/26/2020). It forms an excellent reference. However, a volunteer group continued development into the early 2000’s through the latest Ver. 6.15.05’ dated 3/28/2018.

This guide was originally written for the 1995 release and is being updated to the current release. Please send me any discrepancies you find.

This guide assumes you are quite familiar with basics of computer programming such as number representations in computers and that you have at least a modest familiarity with Forth. I began it as I started to use early Win32Forth but needed to understand its choices on implementation specific details such as memory representation, stack width, input parsing, file access, floating point numbers, etc. As I encountered and resolved these issues I've added them to this note. It is more of a reference document that a teaching document.

If you are completely new to Forth, then refer to *A Beginners Guide to Forth*, Dr. Julian V. Noble, 2001, which is an introduction to Forth using Win32Forth. Published on-line at:

<http://galileo.phys.virginia.edu/classes/551.jvn.fall01/primer.htm#code>

The new user can work through that guide and use the examples to build an early familiarity with Forth. The experienced user will find the examples showing the full nature of Win32Forth (W32F).

See the Documentation and Bibliography sections for more Forth based publications.

## On Standardization

The original Win32Forth development ended at about the time the DPANS Forth Standard (Data Processing American National Standard) was being finalized in early 1994. Win32Forth follows the draft Standard accepted soon thereafter. This Standard is furnished with the system as a series of .HTM files in the range: DPANS.htm, DPANS1.htm to DPANS17.htm, DPANSA1.htm to DPANSA17.htm, DPANSA.htm to DPANSF.htm, and 0001.htm to A0009.htm. The primary entry point is DPANS.

## On Improvements

Win32Forth has greatly benefitted from incremental development over the last 20 years. While all of the original functions have been maintained (see Basic Resources) the development team made very valuable improvements including:

* Support mouse scroll wheel.
* Cut/paste/drag of text.
* Execution window split from the Integrated Development Environment (IDE) window for improved workspace.
* Expanded error messages.
* Line numbering in the editor matching error messages.
* Wider scope and functionality for ‘view’ and ‘see’.
* And much more ‘behind the scenes’.

## How To Install

For an overview of the current status of Win32F start at: <http://win32forth.sourceforge.net/>

You may download a choice of recent releases and Tom Zimmer’s original at:  
 <https://sourceforge.net/projects/win32forth/files/>

The most recent (?) distribution is: Win32Forth, Ver 6.15.05 as:  
w32f61505.exe 2018-03-28 8.3 MB

The accompanying ‘readme.txt’

Latest snapshot from SVN version 6.15.05 March 28th, 2018.

Install it outside your C:\Program Files (x86) directory. EG: C:\Forth is a fine location.

Note: If your anti-virus scanner gives a warning, then the best solution is to inform the vendor that you got a false positive.

See:

<https://www.techsupportalert.com/content/how-report-malware-or-false-positives-multiple-antivirus-vendors.htm>  
for more information.

SourceForge should download into your browser’s specified location for downloads. Create a directory such as c:\Forth\ or c:\Win32F\ and move the downloaded program there. Double-click on it to begin the process.

Amazingly, the loader will recompile the complete system from its source code. Accept the choices to place shortcuts on your desktop. Installation will take several minutes. Do not try to interrupt the process; wait for the completion message.

## Your first Forth program

Open the shortcut with the dotted circle ‘4th’ to bring up the Integrated Development Environment, IDE. You may now start to edit your first program. Using the button ‘Directory’ and the button just below and to the left ‘Change drive/folder’ create or select a scratch folder. Then pull down menus ‘file’, ‘new file’, then ‘save file as’ entering hello.f

In the open editor window enter:  
 : hello .” Hello World” ; (mind the spaces!) and then F12

The W32F system window will open followed by the greeting messages. See something like:  
FLOAD 'C:\Projects\Win32F\\hello.f' ok

Your program has been saved and complied. Run it by typing  
 hello <enter> to see: Hello World

Type words to see hello has been added to the dictionary of Forth words. It is now an integral part of the Forth programming language. Type forget hello to discard it.

You are on youf way to your Win32Forth adventure.

W32F supports all the usual Windows conventions: cut and paste, control-C, control-V, cursor selection, click & drag and scroll wheel.

## Basic Resources

Win32Forth provides the richest Forth environment currently available. It very likely will retain that advantage. Some of its features are:

* Integrated Design Environment, IDE.
* Three stacks including floating-point. (And a ‘locals’ stack.)
* Full length names with adjustable case.
* Compiler structure ‘security’ testing.
* Compiled and interpreted conditionals.
* Native 32 bit structure.
* Byte addressed, little-endian memory space.
* Dictionary segmented into code, system and application.
* Support of DOS commands.
* Floating number support in three formats.
* Trig and transcendentals support.
* N-way hashed vocabularies for search speed. ONLY & ALSO
* Object oriented programming.
* Windows constants and linkage.

## Interacting With The Outer (Text) Interpreter

Upon input, text words and numbers must be separated by a blank or ‘enter’. Forth has two interpreters which execute the internal code corresponding to these words (commands).

First is the 'outer interpreter.' It accepts the text you type from the keyboard (or is read from a Forth file), looks up each word (Forth command) in its dictionary and executes the corresponding internal code. If the dictionary name corresponding to the input word is not found, the outer interpreter tries to convert that text to a number which is then placed on the parameter (numeric) stack. If conversion fails an error message is issued. This same interpreter handles compiling which will be discussed later.

There is no prompt. A correct entry will be followed by an confirming OK. Otherwise you will see a modest error message. For example, type the word ‘drop’ several times and 'enter'. You will get the error message with the offending word is repeated after “Error:’.

drop drop drop <enter>

^^^^

Error(-4): DROP stack underflow

The standard error number is referenced, the offending text and explanation and the offending work underlined ^^^^. The above is for V 6.15.04. Earlier versions do not show the Error Code.

## Inner Interpreter

For completeness I'll mention Forth has an 'inner interpreter' which sequentially scans compiled numeric values in memory and executes the corresponding machine code. This interpretation may nest several levels deep as one compiled Forth word (command) calls others. At the bottom-most level, native Pentium processor machine code is executed. This Forth uses indirect threaded code.

## Some Words To Try

WORDS Display the words in the Current (most recent) vocabulary.

KWORDS Display words in the context vocabulary with their hex addresses.

.WORDS Display the total number of words in the system.

.VERSION Display the version number of Win32Forth.

.CVERSION Display the time and date this system was generated.

.DATE Display the current date.

.TIME Display the current time.

SEE <word> Disassemble and display the following word.

## Documentation

Forth has been documented by a wide range of books and articles. As most of its support developed in the 1980s and 1990s most will be out of print. Checking on Amazon Books and eBay are starting points.

The two introductory classics are Leo Brodie’s Starting Forth, and Thinking Forth. They cover the most common words and operators. They do not go into the advanced forms such as CREATE DOES> (developed later), and our current vocabulary usage. Also, Forth programs often use precise, direct memory manipulation, not covered in those books. And for good reason. . . memory manipulation is very implementation sensitive and risks corrupting the Forth system itself. Available on Amazon Books. A pdf version is posted at: <https://www.forth.com/starting-forth/>

The most useful book is Forth Programmer’s Handbook, 3rd ed., Edward K. Conlkin & Elizabeth D. Rather 2007, pub Forth Inc. Hawthorne, CA. Get it and prize it. It can be found on Amazon Books.

Juergen Pintaske has compiled a very large library of Forth publications available on Amazon. At the Amazon site search on ‘Pintaske’. One in this series is a reference on the origins and philosophy of Forth from the compiled works of its inventor: Charles H. Moore. The volume is Programming A Problem Oriented Language, Forth – How the internals work, 2018 byExMark.

## Using Files

Most programming is accomplished via text files. They allow you to enter words, compile, test and edit. W32F files must have the extent (end in) ‘f’” such as ‘testing.f’. In the IDE, select the Directory and Change Drive/Folder icons to locate your desired working directory. Use File, New File to begin. **Note:** W32F does not accommodate directories with spaces such as ‘c:\test\my demo\’. Rename that with ‘my\_demo’ or something similar.

Customization and file access

.PROGRAM  
.FPATH show present path  
FPATH+ append a path

FLOAD

INCLUDE  
(and more)

.program C:\Forth\Win32Forth-new\Win32Forth\Win32For.exe ok

.fpath  
Base path:  
C:\Forth\Win32Forth-new\Win32Forth  
C:\Forth\Win32Forth-new\Win32Forth  
C:\Forth\Win32Forth-new\Win32Forth

Search path:   
.  
SRC  
SRC\LIB  
SRC\GDI  
SRC\TOOLS  
SRC\RES  
SRC\CONSOLE  
DEMOS  
HELP  
HELP\HTML ok

After editing a file, hit F12 to compile and execute the contents of the file. W32F will open. If the file has compiled without error see ‘ok’. Otherwise you’ll see an error report in the form;  
 2 pick{{ 0 5 pick }}F@ \ top dividend from row 0  
 ^^^^^^  
Error(-13): PICK{{ is undefined in file C:\TEXT\PROJECTS\TALKS\TALKS\_2020\FIG\_PROGRAMMING\_CHALLENGE\MATRIX\ESCHELON.F at line 18

The ^^^^ marks the unknown text with the remaining message locating it. Other common error messages are:  
drop drop drop  
^^^^  
Error(-4): DROP stack underflow in file at line 21

See Section Error Codes for a list of error messages by number.

From either a file or from the console you may load (compile) a file. Enter  
FLOAD file\_name.f or else  
INCLUDE file\_name.f

\*\* here about file location and full path expression.

You my use PRINT file\_name.f to print the file. \*\* Plus there are several options on paging. Note: my W32F under Windows 10 does not print correctly from either the command line or within IDE. I copy my text files to another editing program to print.

MARKER and ANEW

MARKER <word> adds <word> to the dictionary when <word> is executed \*\*??? ANEW <word> places <word> in the dictionary. When the sequence ANEW <word> is later executed the dictionary is trimmed back to that point. This sequence is usually used at the beginning of code under development so recompilation avoids redefinitions.

## Getting Familiar

The documents noted above have a step by step introduction to using Forth. You’ll learn how to execute simple commands from the keyboard and construct words (commands, functions) of your own. Below are several demonstration words to illustrate the power of Forth. They are rather advanced to show some unusual aspects of Forth. Don’t be put off as they dependent on skills and knowledge you will develop over time.

The following words each print 26 letters of the alphabet in alternating case. This will illustrate number of methods appropriate to Forth: 1) using the numeric values of ASCII letter as loop limits, 2) manipulating the bits of a letter to adjust its case, 3) maintaining a value on the stack for repeated use, then dropping it and 4) string input, storage and playback methods. Each example defines a forth work which, when executed, performs the operation.

The word is the whole program. In some languages it would be considered to be or executed from a file. For Forth, after creating the definition, you simply type its name (the part just following the colon ‘:’) at you terminal.

The first example simply ‘plays back’ the string imbedded in the definition or A-1. This is the brute force approach.

: A-1 .” AbCdEfGhIjKlMnOpQrStUvWxUz” ;

The next uses a do loop to print pairs of characters. However, the second letter in each pair has its high order bit set thus generating the ASCII lower case equivalent. This one of ‘tricky’ approaches common in Forth. The 2 +LOOP increments the loop by two on each pass.

: A-2 [char] Z 1 + [char] A   
 DO i emit i 1+ $20 or ( to lower case) emit 2 +LOOP ;

The following is pretty crazy but pretty Forth like. The uses a conditional (IF THEN) nested a loop **(**DO LOOP). It starts by calculating the ASCII offset pattern between upper and lower case characters (‘a’ ‘A” -) which is preserved on the stack over the entire operation. It then uses the numerical values of ‘Z’ 1+ ‘A’ as starting and ending values of loop. Within the loop, each character (the ‘i’) is tested as either odd or even. If odd, the second stack value, the ASCII offset preserved on the stack, is ored to shift the loop value to the equivalent lower case symbol and emit displays it. The final ‘drop’ removes the ASCII upper case shift value.

\*\*\*\* test these for [char]

: A-3 ( use a nested conditional in the loop )  
 [char] a [char] A - ( offset resides on the stack)  
 [char] Z 1+ [char] A (limits across the alphabet)  
 DO i dup 1 and 0= ( test the low order bit)  
 IF over or THEN (capital to lower case) emit  
 LOOP drop ;

The last example shows string processing technique. CREATE A-z 28 allot reserves 28 bites in memory with the name A-z. The S” AbCd. . . z” places the enclosed string in temporary storage and yields its temporary memory address and character count. A-z place moves it into the reserved, named memory space. From a file or from the console, executing A-z A-4 gets the address of the string and prints it using count type. Again, very Forth-like but cryptic to the traditional programmer.

CREATE A-z 28 allot S” AbCdEfGhIjKlMnOpQrStUvWxYz”  
 A-z place ( moves string into allotted space)

: A-4 ( addr --- print alphabet from an array address)  
 count type ;

A-z A-4  
( A-4 executes from the console to display contents of A-z)

## The Stack

All Forth systems have at least two numeric stacks: the Parameter Stack, usually called ‘the stack”, and the Return Stack used for execution control. In addition, Win32Forth has a floating-point stack. More about the latter two stacks later. Our discussion will focus on the Parameter Stack, simply ‘the stack’.

As noted above, if your input text does not correspond to a Forth word already in the dictionary the text will be converted into a number, if possible. That number is placed on the numeric parameter stack (again, simply called the ‘stack’). Each entry is held as a 32 bits, twos-complement integer. In hex, the 32 bit values can run from 00000000 to FFFFFFFF. In decimal they are -2147483648 to 2147483647. Later we'll see Forth also supports 64 bit integers and 80 bit floating point numbers. Stack values may be treated as signed or unsigned.

The Forth word ‘.’ (the ASCII period without the single-quotes, pronounced 'dot') will display the top number from the stack and then discard it. Place several numbers on the stack and then display them. Note they show in the reverse order, with the latest value being displayed first. Try:

1 2 3 4 5 . . . . . 'enter' Remember, you must have a space (or ‘enter’) after each Forth word. You should see 5 4 3 2 1 ok

Win32Forth notes the quantity of stack entries by the number of periods (dots) following the 'ok'. Try this: 1 2 3 'enter' and see: ok . . . with the ‘ok’ followed by three dots. Then enter three dots (to display the three numbers):

. . . 'enter' and see: 3 2 1 ok Now the OK has no following dots as the stack is empty! This simple display will reveal if you have inadvertently left numbers on the stack.

To non-destructively view the numbers on the stack use .s Try this:

1 2 3 .s 'enter' and see: [3] 1 2 3 ok . . . The [3] verifies three values are on the stack which are then displayed bottom of stack to top. Finally the ok . . . with its three dots verifies three values remain on the stack.

To clear the stack simply enter QUIT or some junk text such as ggggg ‘enter’.

## Words and Stack Notation

Each Forth command/function is called a ‘word.’ The words comprising the system appear in the ‘dictionary’ which is searched whenever a command word is executed from the console. A word name can be formed of any printable ASCII text except a blank (ASCII 20).

For documentation it is common to display a Forth word’s operation in the form:

<word-name> ( n1 n2 --- n3 ) The three dashes represent the execution of <word-name> which means n1 and n2 are passed to the word to the data stack and n3 remains on the stack after execution. Thus the notation for ! (store) would be:

! ( n1 adr1 --- ) In this case ! (store) stores n1 at memory address adr1 and leaves nothing on the stack. For @ (fetch) the notation would be:

@ ( adr1 --- n1). The action for @ (fetch) is: memory address ‘adr1’ is on the stack as @ is executed. The contents of memory address adr1 remains on the stack after execution.

? ( adr1 --- ). The action for ? (query) is: memory address ‘adr1’ is on the stack as ? is executed. The contents of memory address adr1 is fetched and displayed. Finally, ? leaves no values on the stack.

In our examples, I will often combine the stack notation and the word name into a single line in the form:

(n1 n2 2DUP d1 n2 n1 n2) 2DUP duplicates the top two 32 bit values on the stack. It also can be considered as duplicating the ‘one’ stack value if it is a double number (64 bits). Another definition could be:

(d1 2DUP d1 d1) duplicate the double number d1.

## Stack Manipulation

DROP ( n --- ) Delete the top value on the numeric stack.

DUP ( n --- n n ) Duplicate the top value on the stack.

SWAP ( n1 n2 --- n2 n1 ) Exchange to top two values on the stack.

OVER ( n1 n2 –-- n1 n2 n1 ) Copy the second value on the stack making it the top value.

TUCK ( n1 n2 --- n2 n1 n2 ) Copy the top value on the stack making it the new third value.

NIP ( n1 n2 --- n2 ) Delete the second value on the stack.

ROT ( n1 n2 n3 --- n2 n3 n1 ) Bring the third value on the stack to the top.

-ROT ( n1 n2 n3 --- n3 n1 n2 ) Move the top stack value becoming the third from the top.

PICK ( nm ... n2 n1 k -- nm ... n2 n1 n[k] ) Using the top stack value k, copy the k-th entry to the top stack value. The initial top of stack value (n1) is at position k=0 (thus not counting k).

ROLL ( nm ... n2 n1 k -- nm ... n2 n1 n[k] ) Using the top stack value k, move the k-th entry to the top of the stack. The initial top of stack value (n1) is at position k=0 (thus not counting k).

?DUP ( n --- [n] n ) If n is not zero duplicate it. Often used before a conditional test such as IF or WHILE.

## 64 bit Stack Manipulation

WIN32FORTH supports 32 bit and 64 bit integers. Here are some of the stack manipulation words. Later we’ll see the 64 bit math operations.

2DUP ( n1 n2 — n1 n2 n1 n2) Duplicate two 32 bit values (or one double [d1 – d1 d1] 64 bit value) to the top the stack.

2DROP ( n1 n2 — n1 n2 n1 n2) Duplicate two 32 bit values (or one double [d1 – d1 d1] 64 bit value) to the top the stack.

2SWAP ( n1 n2 n3 n4 – n3 n4 n1 n2 ) Exchange two pairs of two 32 bit values (or two double [d1 d2– d2 d1] 64 bit value).

2OVER ( n1 n2 n3 n4 – n1 n2 n3 n4 n1 n2 ) Copy the second pair of two 32 bit values (or one double [d1 d2– d1 d2 d1] 64 bit values) to the top of the stack.

2ROT ( n1 n2 n3 n4 n5 n6 – n2 n4 n5 n6 n 1 n2 ) Move the third pair of two 32 bit values (or one double [d1 d2 d3– d2 d1 d3] 64 bit values) to the top of the stack.

2NIP ( n1 n2 n3 n4 — n3 n4) Delete the second-most pair of 32 bit values (or the second double [d1 d2 – d2] 64 bit value) from the stack.

3DUP ( n1 n2 n3 – n1 n2 n3 n1 n2 n3 ) Copy the topmost three 32 bit values to the top of the stack.

3DROP ( n1 n2 n3 – ) Remove the topmost three 32 bit values from the top of the stack.

4Dup ( n1 n2 n3 n4 – n1 n2 n3 n4 n1 n2 n3 n4 ) Copy the topmost four 32 bit values (or two double [d1 d2 -- d1 d2 d1 d2] 64 bit values) to the top of the stack.

4DROP ( n1 n2 n3 n4 – ) Remove the topmost four 32 bit values (or two double [d1 d2 -- ] 64 bit values) from the top of the stack.

## Floating Point Stack Operations

WIN32FORTH offers a rich suite of floating-point operators. Most are named with an ‘F” prefix and execute similarly to their equivalent integer word. Most of these directly address the CPUs floating-point processor, so they execute at full computer speed.

FDEPTH >FLOAT PRECISTION SET-PRECISION  
FARIABLE FCONSTANT FVALUE FTO  
F. FE. FS. F.S F# ##.##e0 (floating input)   
S>F F>S D>F F>D 2F>D FS>DS SFS>DS  
F= F< F> F<= F>= F0= F0> F0<  
F+ F- F\* F/ FNEGATE 1/F F2\* F2/  
FMAX FMIN FABS FLOOR FCEIL FTRUNC FROUND  
F@ SF@ DR@ F! SF! F+! F,  
FDUP FDROP FSWAP FOVER FROT FPICK FNIP  
F2DROP D2SWAP F2NIP  
FPI F0.0 F1.0 F2.0 FBIG FEPS FSMALL  
FSIN FCOS FTAN FSINCOS  
FASIN FACOS FATAN FATAN2  
FASINH FACOSH FATANH   
F^2 FSQRT FLN FLNPI FLOG FALOG FEXP FEXPMI F\*\*  
FL2T FL2E FLOG2 FLN2

## Return Stack Operations

R> ( -- n ) Move the top data stack value to the return stack.

>R ( n -- ) Move the most accessible return stack value to the data stack.

R@ ( -- n ) Copy the most accessible return stack value to the data stock.

A word of caution. While the return stack is often used for the temporary parking of numeric values, additions and removals must be done within a word structure and within any looping control structure as these dynamically use the return stack.

## Input Formats and Number Bases

Values may be input and output in a variety of numeric bases as specified by the value in the variable BASE. Words which set the numeric base: DECIMAL, HEX, OCTAL, BINARY.

You may set the base to any value from 2 to 36 by storing a value into the variable BASE.  
 DECIMAL 2 BASE ! 1111 DECIMAL . <enter> and see 15

For convenience, several input formats are provided to input numbers using specific numeric bases to save having to set and reset BASE. These formats and their size on their respective stacks are: (Note: some of these conversions may only be available in later updates to Win32Forth.)

By the current base: 123 -123 an integer 32 bits

By the current base: 123. –123. a double integer 64 bits

Decimal: &100 -&100 or &-100 32 bits

?? Decimal Double: 124567890. –1234567890. 64 bits

Hexadecimal: $123 -$123 or $-123 0x1234 0x1234L 32 bits

Hexadecimal Double: $123. 0x123. $-123. 64 bits

Floating Value: 1.234e 1.234e2 1.23e-2 the floating point value is placed on the floating stack as an 64 bit value.

Floating Prefix: F# 1234 or F# 123.456 64 bits

ASCII Character: ‘e’ or ‘&’ the numeric value of an ASCII character is placed on the stack as a 32 bit value.

## Numeric Formats

W32F has three numeric formats: single integer, double integer and floating. Integers may be treated as signed or unsigned. A single integer consists of four bytes, 32 bits using signed twos- complement representation.

The most negative value is:

-1 U2/ 1+ . <cr> and see -2147483648

The most positive value is:

-1 U2/ . <cr> and see 2147483647

Double integers of eight bytes, 64 bits are held as two stack entries with the most significant 32 bits topmost on the stack. A double number is entered as an integer with a trailing decimal point which flags the outer interpreter to place that value on the stack using two 32 bit words.

The largest, unsigned double integer is:

-1. ud. <cr> and see 18446744073709551615

The most positive double integer is:

-1. u2/ d. <cr> and see 9223372036854775807

The most negative double integer is:

-1. u2/ 1. d+ d. and see -9223372036854775808

Integers may be placed on the stack by direct (console) input, calculation or from Forth memory. The numeric base for direct text input is specified by executing one of these words: DECIMAL, HEX, BINARY or OCTAL. The system variable BASE holds the current radix for input and output numeric conversion and is set by those commands. A typical input sequence might be to enter a hexadecimal number, duplicate it, switch to binary, print it, switch to decimal and print it again: hex 1234 dup binary . decimal . <cr> to produce: 1001000110100 4660 ok

Thus, we see some of the interactive power of Forth supporting program design and development in a variety of styles.

Notice you must use the D. word to display a double number from the stack. If you use the simple '.' (dot) a double number would appear as two 32 bit integers. Try

hex 7fffffffffffffff. (7 then 15 f’s, note the ending dot) . . <cr> to see: 7FFFFFFF -1 ok

The 7FFFFFFF is the high order 32 bits of the double number (top of stack) and -1 (FFFFFFFF) is the low order 32 bits, second on the stack. The -1 displayed value results from the usual representation of the two's complement format of all bits being set being negative one.

Forth has a rich collection of words to manipulate and display 32 and 64 bit integers.

## Floating Point Input And Output (x86 format)

W32F uses the internal processor’s floating-point computation following the 64 bit ANSI Floating Point Standard. The numeric range is 2.23 x 10-380 to 1.79 x 10308 (?) with a safe maximum of 16 (see below) digit precision. The floating-point processor chip uses an 80 bit internal format (?).

This 64-bit format uses one bit for the sign of the significand, 11 bits for the exponent field and 53 bits for the significand. ?hidden bit for 64?

Bounds on conversion between decimal and binary are: if a decimal string with at most 16 significant digits is correctly rounded to an 64-bit IEEE 754 binary floating-point value (as on input) then converted back to the same number of significant decimal digits (as for output), then the final string will exactly match the original; while, conversely, if an 64-bit IEEE 754 binary floating-point value is correctly converted and (nearest) rounded to a decimal string with at least 16 significant decimal digits then converted back to binary format it will exactly match the original. These approximations are particularly troublesome when specifying the best value for constants in formulae to high precision, as might be calculated via arbitrary-precision arithmetic.

Use NN set-precision to set the number of significant digits to be accepted or displayed.

The byte count for a float is given by B/FLOAT = 8, the cells per float by CELLS/FLOAT = 2.

Floating point numbers (often called ‘reals’ or ‘real numbers’) are supported on a floating-point stack holding up to 250 entries with each entry of 8 bytes. Numbers may be placed on the floating-point stack by direct numeric input in the following formats:

12345e2 f. 'enter' to produce 1234500 that is 12345 times 10^2.

123.45e0 f. 'enter' to produce 123.45

123.45e3 f. 'enter' to produce 123450.

A very general floating-point input may be obtained without exponential notation by preceding the numeric text by the Forth word F# followed by a space. For example

F# 123.4, F# -123.4 or F# 1234e1

F# is a Forth command word (used as a prefix) which processes the following character string as a floating-point number.

F# 1.2345 is equivalent to 1.2345e0

For floating point output, the total number of digits displayed may be set by:

10 SET-PRECISION

## Display and Printing Formats

The basic numeric output words are:

. Display the top of stack in the current base.

? From an address on the stack, display its (32 bit) contents.

.s Preserve and display the stack values (32 bit).

a .R Display the top of stack in field a digits wide.

a U.R Display the top of stack, unsigned, in a field a digits ide.

a b H.R Display ‘number a’ as hexadecimal a field b characters wide

a b H.N Display ‘number a’ as hexadecimal showing b digits.

U. Display the top of stack as an unsigned integer.

H. Display the top of stack in hex.

B. Display the top of stack in binary.

D. Display the top two stack values as a double number.

a D.R Display a double number in a field a digits wide.

F. Display the top of floating-point stack.

F.S Preserve and display the floating point stack values.

FS. Display the top of floating-point stack, scientific notation

FE. Display the top of floating-point stack in engineering notation.

## Pictured Numeric Output

Forth provided a very flexible set of ‘pictured’ output formatting primitives which allow you to define specific output formats. Pictured output always operates on double numbers. To print the usual signed 32 bit, single precision stack value use S>D. Conversion proceeds from the **least significant digit** to the most significant (right to left). Thus a leading ‘$’ or ‘-‘ appears at the end of the conversion process.

IMPORTANT: These words must be used within a word definition. They use the scratch workspace PAD which also may be used during console input or output. For example:  
 : demo S>D <# # # [CHAR] . HOLD #S [CHAR] $ HOLD #> TYPE ;  
 enter 12345 demo and see $123.45

<# Prepare a print string buffer (located at PAD) to hold the converted text.

# (dn --- dm ) Convert one digit of double number n by the current BASE and append it to the print string buffer. The balance remains as m.

#S (dn --- dzero ) Convert and append all the digits of double number dn into the print string buffer leaving double number dzero.

#> ( dn --- addr count) Close the print string buffer and return its address and character count.

SIGN ( dn --- dn ) Append the sign of n to the print string buffer.

HOLD ( c --- ) Append the ascii text character c to the print string buffer.

[CHAR] Convert and compile the following ASCII charter as a literal. It must be used within a ‘:’ a colon definition. At run-time the ASCII value will be added to the stack. It can used in the form ‘[CHAR] $ HOLD’ , or ‘[CHAR] . HOLD’ to place a symbol into a print string. (When interpreting text use CHAR.)

## Case Sensitivity

In the usual case the Forth outer interpreter converts all input to upper case. Thus 'DUP' 'Dup' and 'dup' are all equivalent when searched for in the dictionary and will match the resident word in the dictionary DUP. The entire resident system was compiled following this convention so viewed words will appear in upper case. Type: WORDS 'enter' to check this.

However, you may force the outer interpreter to retain the original input case by setting the variable CAPS to OFF (false or zero). To specify case sensitivity enter: CAPS OFF 'enter' ; to restore case insensitivity enter CAPS ON 'enter' Note that last sequence must be entered in upper case in order to match the dictionary entries for those two words!

As consequence of this convention, with CAPS ON, new words entered into the dictionary will have their names stored in upper case. Again, if you wish to have mixed case simply turn caps off with: CAPS OFF 'enter'

## More on Forth Words

Forth commands are referred to as 'words' as they have a name and a definition, their runtime execution. Words must be separated by a trailing blank or end of line ('enter'). Likewise, numbers be delimited by a trailing blank or end of line (‘enter’).

A Forth word encountered by the outer (text) interpreter will have its compiled code executed via the inner interpreter.

Forth is an extensible language. By this I mean you may add (compile) words into the dictionary which then form your application. These new words add new capabilities to Forth itself. A typical application grows into many words which are ultimately combined into one word which, when executed, starts the application.

An alternative is to have an application consist of a collection of Forth words (a vocabulary) which are executed interactively from the keyboard as you type them or they are read from a file.

## Tool and Utility Words

You may list the contents of the dictionary with a variety of support words. The space bar will start and stop listing if scrolling off the screen:

WORDS Display the words in the Current (most recent) vocabulary.

VOCS Display the vocabularies available.

ORDER Display the vocabularies in the CURRENT search order and the

CONTEXT vocabulary into which new definitions will be placed.

SEE <word> Disassemble and list the following word.

VIEW <word> Display the source code for the following word.

.S Display and maintain the contents of the data stack

WITH-ADDRESS Used before WORDS, then WORDS will also show the word’s hex memory address.

KWORDS Display words in the context vocabulary with their hex addresses.

.RSTACK Maintain and display the contents of the return stack.

DEPTH . Display the quantity of values on the data stack.

FDEPTH . Display the quantity of values on the floating point stack.

Learning About Your System and Computer

These utility words will reveal information about your current environment:

.WORDS Display the total number of words in the system.

.CUR-FILE Display the name of the current file.

.FILE Display the name of the current file.

.DEFERED Display the name of deferred words. This advanced function allows forward references in Forth.

.PROGRAM Display the path and name of the program being executed.

.FPATH Display the Forth directory search path list.

.LOADED Display all the Forth programs currently loaded.

.PLATFORM Display the Microsoft operating system.

.EDITOR Display the editor, browser, shell and DOS strings.

.BROWSE Display the editor, browser, shell and DOS strings.

.FONTS Display the available Windows fonts.

.FREE .MEM Display the amount of used and available program memory.

.MEM-FREE Display the bytes of available program memory.

.VERSION Display the version number of Win32Forth.

.CVERSION Display the time and date this system was generated.

TIME&DATE Leave on the stack: sec min hour day month year.

.DATE Display the current date.

.TIME Display the current time.

.HELP Display the current help string

.PLATFORM Display the current operating system.

NT? WIN95? WIN32S? Return a boolean for the current operating system.

.RSTACK Display the contents of Forth’s return stack.

.COUNTS Display the thread number and number of words of CONTEXT.

.THREADS Display each thread number and the words in contains.

.USERSIZE Display address of next user-space available and remaining space.

START/STOP When included in a loop, will pause and continue execution upon ‘space-bar’ action.

## Memory Addressing

Forth is unusual in that you have direct access to and often use its actual memory space. This is a memory space dedicated to Forth shared with its registers, two interpreters, data storage, scratch memory and dictionary. The physical memory (ram) of the host computer is accessed under very limited situations.

The Forth Memory Space is arranged as a sequence of eight bit bytes located within the computer's much larger memory space. Win32Forth addresses each byte using a 32 bit memory addresses. Each Forth address occupies four bytes whether on a stack or in memory. The term applied to memory units are: a **byte** (8 bits), a **word** or **half-cell** (2 bytes) and a **cell** (4 bytes). In addressing a sequence of bytes their addresses would be $3400, $3401, $3402, $3403, etc. Addressing cells would have addresses: $3400, $3044, $3408, etc.

The Forth memory environment appears run from $0000 to above $000C0000 (?). When Forth addresses its memory space, a translation is made to your computer's physical memory. Direct access to physical memory is very unusual, limited processes deep within Forth operation.

Although you will rarely, if ever, need to use a physical memory address two operators are provided for this conversion. REL refers to Forth’s relative (internal) addresses; ABS specified a processor absolute address. If you have a Forth address on the stack and execute REL>ABS that value will be translated to the corresponding physical memory address. The Forth word ABS>REL makes the opposite conversion. These must be used with extreme caution.

## Forth’s Addressing Conventions

(A major change to memory allocation were made about 2002. The following should be reviewed and corrected.) Win32Forth supplies about nine (?) megabyte of memory space for applications. The first 700,000 bytes (?) become Forth's application memory space. It holds working Forth programs, the user's applications and the user's data. The Forth components of code fields, parameter fields and xx reside in this low memory space. Upon startup about 245,000 (?) bytes are taken by precompiled code. The user has available about 8,000,000 bytes as displayed by the command .free or its synonym .mem.

Win32Forth also uses memory above C0000h (?) as its System memory space. That space contains the information used for interpretation of user commands, many structured words lists (vocabularies) and associated pointer values. Collectively, these items are called the word headers.

Together, the two address spaces are called the 'dictionary' as it contains lists of words and definitions, that is, the execution process associated with each word. The dictionary is rather like an assembly language program in which the symbol table is preserved at run-time and each program component is executable by name. Or, think of Forth as a high-level language with all program labels preserved and accessible at run time.

In about 2002 a major change was made to the memory allocation methods with pre-allocation of memory space to specific functions. To check memory use and availability enter either .free and .mem. The report will be similar to:

.free

Section Address Total Used Free

------- --------- --------- --------- ---------

\*LOCALS CF1204h 4,096 0 4,096

\*PROCS 41560Ch 49,920 6,134 43,786

CODE 401000h 40,960 12,380 28,580

APP 40B000h 9,003,008 256,836 8,746,172

SYS CA1000h 1,024,000 332,944 691,056

\* areas are inline

Malloc/Pointer: 10,087,444 ok

In the 1996 version of Win32Forth the display was:

Application address: 00000000h  
 Total: 716,800  
 Used: 244,656  
 Free: 472,144  
 System address: 000C0000h  
 Total: 409,600  
 Useid: 191,653  
 Free: 217,947  
Malloc/Pointer Mem: 26,512

## Memory operators

The commonly used words (operators) for accessing memory appear as short symbols: @, !, “,” <comma>, ?, etc. The Forth word @ (pronounced 'fetch') replaces an address on the stack with the contents of that address. The Forth word ! (pronounced 'store') stores the value second on the stack at the address at the top of the stack. Those original values are discarded. The operator , <comma> places the top of stack value into the next available memory location.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Object | Bits | Fetch | Store | Compile |
| Byte | 8 | C@ | C! | C, |
| 1/2 cell or word | 16 | W@ | W! | W, |
| Cell | 32 | @ | ! | , |
| Fetch & sign extend | 16>32 | SW@ |  |  |
| Increment |  |  | +! |  |
| Display cell contents |  |  | ? |  |

## Data Structures

The default data unit in Win23Forth is 32 bits, 4 bytes, its data stack width. Thus constants, variables and values are all of that size.

Data is stored in memory as low byte first, at the low address, often referred to as ‘little-endian’. Thus, in the variable HOT the first byte may be accessed with HOT C@, the first two bytes HOT W@ and the full four bytes with HOT @.

n CONSTANT <name> Create a word that later returns is value.

VARIABLE <name> Create a word that returns in cell storage address. The initial contents of that cell must be assigned later.

CREATE <name> Create a word that returns its address but assigns no memory.

n USER <name> Create a word that returns the storage address specific offset into the user area.

n VALUE <name> Create a word that returns its value when executed. A new value may be assigned by: <n> TO <name>. The value may be incremented by: <n> +TO <value>.

[String Variable] Make a named storage area in memory in the form ‘CREATE string-name’. Then <n> ALLOT to reserve memory. Then S” to create a string. Then move the string into the allocated area. An example:

CREATE SCRATCH 10 ALLOT C” abcdef” SCRATCH PLACE

\*\*\* check the above example for count and use of place \*\*\*

. `

## Constants

You may add named constants to Forth. Just enter: <number> CONSTANT demoC 'enter' A new word named demoC will be created and set equivalent to <number>. Upon later execution, demoC will place its value on the stack. Try:

1234 CONSTANT demoC 'enter' then input demoC . 'enter' and see: 1234 ok

Note, if you try to create a constant without a parameter on the stack you will receive an error message that the stack under flowed. Try:

constant xxx 'enter' and see:

Error: XXX stack underflow The most recent input text will be echoed back along with an indication of the problem.

## Variables

You may add named variables within Forth. Simply enter: VARIABLE <var-name> where var-name is the chosen name for the variable. You must then store an initial value for <var-name>. Win32Forth creates variables with the value of zero but this may not be the case in other Forths. To assign an integer value to a variable: VARIABLE demo 12345 demo ! 'enter'

To see the contents of that variable enter: demo ? 'enter' and see 12335 ok.

To set a variable to zero use: <var-name> OFF.

When executed, a variable leaves its Forth storage address on the stack NOT the value it contains! This access allows you to read and write to that variable by subsequent words. The Forth ? word retrieves the contents stored at an address then displays it. Try BASE ? . All values are 32 bits.

## 2VAR and FVAR?

## VALUE Variable

The data word created by VALUE may be used as a variable that is read often and written to rarely. It is declared with an initial value in the form:

15 VALUE A-VALUE used as: VALUE . ‘enter’ and you see: 15 ok

Upon each later execution of A-VALUE its 32 bit value will placed on the stack. The operator TO is used to assign A-VALUE new 32 bit value in the form:

45 TO A-VALUE try it and see A-VALUE . ‘enter’ 45 ok

To increment a value word use: 2 +TO A-VALUE A-VALUE . ‘enter’ 2 ok

The form: &OF A-VALUE will compile storage address of A-VALUE as a literal.

## Data In Memory

Forth has a rich variety of representations of information in memory. However, Forth is an untyped language. That is, the running program does NOT keep track of the various information formats, where the data may be located and which words may operate on specific data. This compatibility is a design responsibility of the programmer.

As an extreme example you may place an ASCII character on the stack, add an integer to it, mask it with a hex number and convert the result to a floating point number and convert it to the floating arcsin as an angle:

'A' 1 + $F and s>f fasin F. ‘enter’ and see .857072 ok

Note in most cases Win32Forth is handling 32 bit integers or 8 bit characters (i.e. @ ! , (comma) etc.) For 16 bit operations see w@, w! and w, .

## Examples of Data Storage In Memory

The word DUMP allows you to examine any of the memory in Forth’s dedicated address space. The syntax is: ADDR COUNT DUMP . From the address and byte count on the stack DUMP displays the contents of that address range as hex bytes and the equivalent ASCII text. The leading address is displayed in the current numeric base.

The following examples show the use of :

S” Compile a string which will later generate its address and byte count.

C” Compile a string which preceded by its byte count.

CREATE Create a named entry in the dictionary with no allocated memory.

C, Store a byte from the stack into the specified address.

W, Store a word, 16 bits, from the stack into the specified address.

, Store a cell, 32 bits, from the stack into the specified address.

F! Store a floating point number at the given address

2! Store a double number at the given address.

FVARIABLE Create a floating point variable holding 10 bytes.

2VARIABLE Create a variable capable of holding a double number, 64 bits, 8 bytes.

: A S" ABCDEFG" DUMP ; A <cr>

244665 41 42 43 44 45 46 47 ABCDEFG

This sequence executes the word A containing a compiled string and displays that string with its memory location, hex values and ASCII values. Upon execution S” locates its string by its address and byte count which are the values passed to DUMP.

: C C" ABCDEFG" COUNT SWAP 1- SWAP 1+ DUMP ; C

3BBE0 07 41 42 43 44 45 46 47 .ABCDEFG

This sequence executes the word C containing a compiled string, adjusts to locate the string including its initial count byte (the $07) and displays using DUMP. A string with a leading count, located by the address of that string is called a “counted string.”

CREATE D HEX FF C, D 8 DUMP

3BBB4 FF 00 00 00 00 00 00 00 ........

This sequence creates storage named D, places a single hex byte $FF there (using C,) and displays that value and the seven bytes following. Only the $FF byte is part word D.

CREATE F HEX 1234 W, F 8 DUMP

3BBC4 34 12 00 00 00 00 00 00 4.......

This sequence creates storage named F, places the 16 bit hex word $1234 there (using W,) and displays that value and the six bytes following. Only $1234 is part of the word F. This storage method is called ‘little-endian’ as the lower order byte is stored first in memory proceeding to the high order byte.

CREATE G HEX 12345678 , G 8 DUMP

3BBD4 78 56 34 12 00 00 00 00 xV4.....

This sequence creates storage named G, places the 32 bit hex number $12345678 there (using ‘comma’ , ) and displays that value from the lowest order byte to the highest. Only the number $12345678 is part of the word G. Note that the little-endian nature continues here. The storage space taken is 1 cell or 4 bytes.

CREATE H HEX 12345678 W, H 8 DUMP

3BBD4 78 56 00 00 00 00 00 00 xV4.....

This sequence creates storage named H, places the lower 16 bits of the 32 bit hex number $12345678 there (using ‘Wcomma’ , ) with the upper 16 bits being discarded. It then displays that value from the lowest order byte to the highest. The storage space taken is 1 word or 2 bytes.

FVARIABLE I 8 S>F I F! I 10 DUMP

244660 00 00 00 00 00 00 00 80 02 40 .......€.@

This sequence creates a floating point variable named I which can hold a 10 byte, 80 bit floating point number, converts the integer 8 to the equivalent floating point number, stores that number in H and finally displays the 10 bytes holding the floating point result.

2VARIABLE M HEX 0123456789ABCDEF. M 2! M 8 DUMP

3BBB4 67 45 23 01 EF CD AB 89 gE#.ïÍ«‰ ok

This sequence creates space for two cells of storage named M, stores the 64 bit hex double number $012345678ABCDEF there (using 2!) and displays that value from the lowest order byte to the highest. Note that the double number ends in a ‘.’ (dot) forcing input conversion to a double number and a double number uses two 32 bit stack values. The low order 32 bits (second on the stack) is stored at the low address ($3BBB4) starting with its low order byte using four bytes, one cell. The high order 32 bits (top of stack) is stored, also low byte first, in the following four bytes.

## Word Types

Defining words for execution: CODE : CREATE

Defining words for data structures: CONSTANT, VARIABLE, VALUE

Flow control, conditionals: IF THEN BEGIN UNTIL

Compound words: DOES> ;CODE and DEFER

Assembler words: ADIW, IF,

Label optional, may be used to label un-named assembly code sequences

## Text Strings

Text strings may be interpreted or compiled with:

**.”** text”Compile a string into the dictionary (at HERE) ending with a zero byte. Upon later execution “text” will be displayed.

C” text” Compile or execute a counted string that will yield the address of its leading count upon execution.

S” text” Compile or execute a string that will yield its address and count upon later execution.

.( text) When placed in source code, as that source code is compiled “text” will be displayed. Usually used to add commentary or check points as source code is compiled.

ABORT” msg” Upon execution, if the input stack parameter is non-zero display “msg” and abort. This is primarily used to exit upon an error condition.

## String Utility Words

COUNT From the address of a counted string, place the count on the stack and return the address of the (next) first character. Typically used as ‘COUNT TYPE’.

CHAR accept the next non-blank character in the input stream

[CHAR] compile the next non-blank character as a literal

WCOUNT

LCOUNT

“CLIP”

PLACE

+PLACE

-TRAILCHARS

-TRAILING

-NULLS

/STRING

W32F has a rich selection of words to support text strings, in reality, any sequence of bytes in memory valued 0..255. Note the bytes of a 16 or 32 bit numeric value are stored in low-high order. WSKIP, LSKIP, WSCAN and LSCAN operate over two or four bytes in memory in high-low order. Thus these words are more useful to locate a sequence of text characters rather than numeric values.

adr1 len1 char SCAN adr2 len2

Examine the byte sequence starting at adr1 over len1 bytes. Return the address adr2 of the first byte matching ‘char’ and length len2 of the remainder of the byte sequence. The byte sequence may represent text or any sequence of byte values 0..255.

adr1 len1 word WSCAN adr2 len2

Examine the byte sequence at adr1 over len1 bytes. Skip over leading occurrences matching the integer word value, a 16 bit value, returning the address adr2 of the first byte after the 16 bit value, high-low ordered, and length len2 of the remaining byte sequence. The byte sequence may represent text or any sequence of bytes 0..255. Note that the bytes of a 16 bit numeric value are stored in low-high order. However WSKIP makes its selection over each two bytes in memory in high-low order.

adr1 len1 long LSCAN adr2 len2

LSKIP has an action similar to WSKIP except long is a 32 bit value. Likewise LSKIP looks for a four byte sequence in order from highest byte to lowest byte. Thus it is appropriate to skip over a four byte sequence text characters.

adr1 len1 char SKIP adr2 len2

Examine the byte sequence starting at adr1 over len1 bytes. Skip over leading occurrences matching char returning the address adr2 of the first byte that is not char and then the length len2 of the remainder of the byte sequence. The byte sequence may represent text or any sequence of byte values 0..255.

adr1 len1 word WSKIP adr2 len2

Examine the byte sequence at adr1 over len1 bytes. Skip over leading occurrences matching ‘word’, a 16 bit value, returning the address adr2 of the first byte after the occurrence of the 16 bit value (two bytes) ‘word’ and length len2 of the remainder of the byte sequence. The byte sequence may represent text or any sequence of bytes 0..255. WSKIP makes its selection over each two bytes in memory in high-low order. If SKIP was examining the character sequence: ‘abababxxxx’ for ‘ab’ it would skip over the leading ‘ababab’ and return the address and count for ‘xxxx’.

adr1 len1 long LSKIP adr2 len2

LSKIP has a similar action as WSKIP except long is a 32 bit value. LSKIP looks for a four byte sequence matching in order from leading byte to trailing byte. If SKIP was examining the character sequence: ‘abcdabdcxxxx’ for ‘abcd’ it would skip over the leading ‘abcdabcd’ and return the address and count for ‘xxxx’.

adr1 len1 char -SCAN adr2 len2

Examine the byte sequence ending at adr1 over len1 bytes starting a high memory (adr1) working down toward low memory. Return the address adr2 of the first byte ‘char’ and length len2 of the remainder of the byte sequence. The byte sequence may represent text or any sequence of byte values 0..255. –SCAN searching for ‘d’ over ‘abcdefgh’ would return the address for character ‘d’ and length for ‘abcd’.

adr1 len1 char -SKIP adr2 len2

Examine the byte sequence ending at adr1 over len1 bytes starting a high memory (adr1) working down toward low memory. Return the address adr2 of the first byte that is not ‘char’ and length len2 of the remainder of the byte sequence. The byte sequence may represent text or any sequence of byte values 0..255. –SKIP searching for ‘f’ over ‘abcdeffffff’ would return the address of character ‘e’ and length for ‘abcde’.

adr1 len1 adr2 len2 COMPARE n

Compare a byte sequence starting at address adr1 of length len1 to a byte sequence at adr2 of length len2. Return the value n=0 if sequence 1 matches sequence 2 byte for byte. Return the value= –1 if sequence 1 is less than sequence 1. Return the value n=1 if sequence 1 is greater than sequence 2. The byte sequence with a larger length is declared larger. If len1=len2 then the bytes are compared in order between to two sequences and the sequence with the earliest higher byte value is declared as larger. The bytes may be any value 0..255.

adr1 len1 adr2 len2 SEARCH adr3 len3 flag

Search over a byte sequence starting at address adr1 of length len1 for a byte sequence of adr2 of length len2. If a match is found return the address adr3 of the first matching byte, the remaining byte count in the sequence len3 and a true flag –1, If no match is found return the original adr1, len1 and a false flag zero.

Sorting and Numeric Comparison

W32F provides several utility words useful as the basis for a more comprehensive application word set:

a1 n1 LARGEST a2 n2

Over a sequence of 32 bit numeric values (low byte to high byte) starting at byte address a1 over n1 4 byte cells (an array of n1\*4 bytes) return the address of the largest numeric value.

a1 n1 CELL-SORT

Sort in place a sequence of 4 byte, 32 bit cells starting at address a1 over n1 cells (an array of n1\*4 bytes). The resulting sequence will be in low to high order.

a2 n2 BYTE-SORT

Sort in place a sequence of bytes starting at address a1 over n1 bytes. The resulting sequence will be in low to high order.

## About File Names

Win32Forth expects source code files to have the extent .F. However, as this .F is not automatically appended by the Forth editor, you must include it and save such files with a name in the form FileName.F In addition a file may not have spaces in its name. Thus NEW-PROJECT.F is an appropriate file name; NEW PROJECT.F is not.

## The Editor

Win32F contains WinView functioning as a visual editor (cursor controlled, What You See Is What You Get) and as a command driven editor using keyboard keys in combination with the Control and Shift keys. Its source code is located in WinView.F

[Xxx?] will open a new file and begin editing it. Editing of an existing file may be started by XX.

The two most important keys are F 12 to save and load the current file and Cont L to load the current file without saving. Common commands are Cont c to copy selected text to Windows clipboard, Cont x to cut the selected text and save it to the Windows clipboard and Cont v to insert that text at the cursor position. Cont f will open a search window.

## Commenting Text

W32F provides several methods to add comments to source code, usually in files. The comment forms skip the enclosed test. Descriptive comments of the code functions are essential. You or someone else well return to your code years later and need all the guidance possible. Comments are most useful to the describe the logic or purpose, not the exact operation. For example, “compute the circumference” is better than “multiply radius squared by pi ”.

( text) Skip ‘text’ until the closing ‘)’. Both ‘( )’ must be on the same line (before the invisible end-of-line character).

(( text)) Skip ‘text’ until the closed ‘))’. The closing ‘))’ may be on subsequent lines. This is often used to skip over extensive comments.

.( text) Skip ‘text’ until the enclosing ‘)’ and display ‘text’ when compiling. Usually used to add narrative or checkpoints within source code files.

\s Skip the following text until the end of file. This form is often used during code development to stop compilation before untested code. It may also mark the beginning of extensive commentary.

## Keyboard Shortcuts

While most editing is done with the mouse and cursor, a very rich selection of keyboard input enhance the editing process.

Cont A Highlight all the text in the current file.

Cont B debug-word

Cont shift B debug-word

Cont C Copy the highlighted text to the clipboard.

Cont Shift C >F

Cont E debug-buttons

Cont Shift E expand-tabs

Cont F Open a search and replace window.

Cont Shift F find-in-files

Cont G delete-character

Cont Shift H make-hex

Cont L load-active-file

Cont Shift M replay-macro

Cont N new-text

Cont O Open a file for editing by its name.

Cont Shift O Open a file for editing by it highlighted text.

Cont P Open Windows print dialog to print the current file.

Cont Shift P Open WinFiew’s printing option window.

Cont Q highlight-mark

Cont R reformat-text

Cont Shift R WinViewWindow repeat-amacro

Cont S save-text

Cont Shift S Start/Stop-macro

Cont T word-delete

Cont Shfit T word-undelete

Cont U highlight-case-toggle

Cont V Paste text from the clipboard

Cont Shift V Paste-date/time

Cont W close-text

Cont Shift W close-all-text

Cont X cut-text

Cont Y line-delete

Cont Z word-undelete

F1 s" STARTUP.TXT" "browse

F2 Open the help file.

F3 find-text-again

F4 back-find-text-again

Cont F3 find-text-highlight

Cont Shift F3 replace-text

F5 replay-macro

F9 hyper-link

Cont F9 next-hyper-link

Shift F9 browse-toggle

Cont Shift F9 word-link

F10 close-text

Shift F10 Save all open files and exit the editor.

F11 temp-text

Cont Shift F11 do-html-link

Cont Shift F12 save-text-pc

F12 Save the current file and load it.

DownV Move down one row of text.

Cont DownV 1 +row-scroll

Shift DownV highlight-down

End Move to the end of the current line.

Shift End Highlight text from the cursor to end of the current line.

Cont Shift End Hilight the entire line holding the cursor.

Cont End Movd to the end of this document.

Home home-line

Shift Home highlight-home-line

Cont Home home-doc

Left Arrow Skip one character to the left.

Cont <Left Skip to the start of the next word on the left.

Shift <Left Highlight one character to the left.

PgDn 1 +page-cursor

Cont PgDn next-link

PgUp -1 +page-cursor

Cont PgUP prev-link

Right Arrow Move one character to the right.

Cont Right> Move one word to the right.

Shift Right> Hilight one charter to the right.

Up Arrow Move up one row.

Shift Up -1 +row-scroll

Shift Up highlight-up

Tab insert-tab

Shfit Tab back-tab

Insert ( ignore insert )

#### Shift Insert paste-text

Cont Insert copy-text

Delete Delete the character to the right.

Shift Delete cut-text

Cont Delete word-delete

Backspace Backspace to the left deleting the character.

Shift Backspace next-window

Esc no action

LF (control enter) goto-line

Enter (cr) do-cr

Otherwise insert the typed character into the file at the cursor.

## User Support words

Forth has a variety of words for user help and to support program documentation. We have already used the word WORDS . Try it again now as: WORDS 'enter' and see the four column display of dictionary words. You may start and stop the display with the space bar and terminate the display with 'escape' . If WORDS is followed by text, only the dictionary words containing that text will be displayed. Try WORDS / ‘enter’ to see all the words doing division.

WORDS Display all words in the dictionary.

WORDS <text> Display the words in the dictionary containing ‘text’

SEE <word> Disassemble a word from the dictionary (memory).

LOCATE <word> Show text from source code file

VIEW <word> Locate the source code text of the following Forth word (from a file) and display in the Editor's window.

EDIT <file> Edit a file by name

BROWSE <file> View a file but protect from editing

FTYPE <file> Copy a file to the console screen, displaying it

COPYFILE <file>

FORGET <word> Remove <word> and all subsequent words from the dictionary.

EMPTY FORGET all words the user has added to the dictionary.

ANEW <word> Trim the dictionary back to this starting point and

create <word>. Later execution of <word> will remove it

and all subsequent words from the dictionary.

QUIT Empty the stacks and perform a warm start.

## Comments

When you prepare a file of Forth source code you may enclose comments in parenthesis in the form: ( This is a line of comments.) Note that the leading left parenthesis must have a following blank as it is a Forth word, will be looked up in the dictionary and executed. It will step over the following text on the same line until the closing right parenthesis ‘)’ .

The word ‘\’ will skip over all of the remaining text on that one line. The word ‘((‘ will skip over multiple lines of text until finding the mating word ‘))’ .

Finally, you may include text to print during compilation in a source code file with the word ‘.(‘ (pronounced dot-paren). It will display on the console screen the following text until a closing right parenthesis ‘)’ . An example is: .( We have reached line 200 of the source file.)

\ Skip text until the end of the current line.

(( )) Skip text until the closing ))

.( ) Display and skip text until the closing )

false [IF] . . . [THEN] Skip the text between [IF] and [THEN] over all the intervening lines. Used for large blocks of comments as well as Conditional Compiling, see below.

\s Skip all text until the end of file.

## Cell Size and Manipulation

In this Forth the basic unit of addressing is the byte (8 bits). To accommodate the full memory range a machine and Forth address occupies 4 bytes or 32 bits. This 4 byte unit of memory called a ‘cell.’

This means any Forth address stored in memory and most numeric data fields are 4 bytes wide. Floating point numbers and string have their specific allocation size. Consistent with this cell size, variables, constants, fetching and storing most often involved a 4 byte cell.

The allocation of memory is done by n ALLOT from a byte quantity ‘n’. To facilitate allocating memory and moving across data fields a rich set of words are provided. The word CELLS converts a cell quantity to byte quantity suitable for memory allocation. If you wanted storage for five integers the sequence 5 cells allot would generate 20, the number of bytes, and allocate 20 bytes in memory.

CREATE A-STRUCTURE \ makes a dictionary entry for the storage

5 CELLS ALLOT \ calculates 20 and allocates that storage space in memory.

Words to manipulate cells:

CELLS ( n1 -- n1\*cell ) convert a cell quantity to the equivalent byte quantity

CELLS+ ( a1 n1 -- a1+n1\*cell ) multiply n1 by the cell size and add the result to address a1  
CELL+ ( a1 -- a1+cell ) increment an address by cell size (4 bytes)

CELL- ( a1 -- a1-cell ) decrement an address by cell size (4 bytes)

+CELLS ( n1 a1 -- n1\*cell+a1 ) convert n1 to the equivalent byte count and add the result to address a1

-CELLS ( n1 a1 -- a1-n1\*cell ) convert n1 to the equivalent byte count and subtract result from address a1

## Reading and writing memory

Memory is arranged as a sequence bytes. Each byte consists of 8 bits; two sequential bytes form a half-cell or ‘word’ of memory (not to be confused with a Forth word in the dictionary); four sequential bytes form a ‘cell.’ A number is placed into memory low byte first (‘little-endian’). Thus if the hex number $12345678 is stored in memory it will appear as this hex byte sequence: 78 56 34 12

To see memory contents and its ASCII equivalent use DUMP. DUMP requires a Forth memory address and byte count then DUMP. Try: 1000 15 dump 'enter' and you will see:

1000 30 5E 0E 00 3C D6 0E 00 48 D7 0E 00 18 89 0E 0^..<Ö..H×...‰. ok

To examine the execution portion of a Forth word input: ' DUMP 16 DUMP 'enter' You will see:

31356 60 00 00 00 E4 08 00 00 44 7A 00 00 B8 08 00 00 `...ä...Dz..¸... ok

' DUMP returns the address of DUMP's code field, 31356. Its contents are 60 00 00 00 (remember the low to high byte order, so the number is 00000060h) which is a pointer to the common execution code for a word which was defined by the Forth word : (colon) as DUMP was. As an aside, that execution code fragment is named (DOCOL).

After the code field, the following 4 byte sequences are the compiled addresses of the Forth words which are interpreted when DUMP is executed.

In this issue of Win32Forth the following values contained within a code field specify operation of that word. These values would likely change in other releases.

DOCON A0 00 00 00 The runtime execution for a constant

DOVAR B8 00 00 00 The runtime execution for a variable

DOCOL 60 00 00 00 The runtime execution for a colon definition

; 10 06 00 00 The runtime execution for EXIT

DODOES The runtime execution for a constant

DOUSER The runtime execution for a constant

DODEFER The runtime execution for a deferred word

DOVALUE The runtime execution for a value

DOVALUE! The runtime execution for word to rewrite a value

DOVALUE+! The runtime execution for word to increment a value

DO2VALUE The runtime execution for a double value

## Locating Words In The Dictionary

The Forth word ' (the single quote, pronounced tick) accepts the following word ending in a blank or <return> and searches for it in the dictionary, returning the code field address. A variety of words facilitate moving from one field to another within the header and parameter portions of the word. A forward link exists within the word’s header pointing to its cfa field located in its parameter field. To move from that code cfa field, in the body of the word, to its header the word >NAME searches over all words in the dictionary.

' <word> --- cfa tick returns the code field address

FIND addr --- xt or addr --- addr 0 from the counted string at addr return the execution token xt if found. If not found return addr and zero.

>BODY cfa --- pfa code field address into parameter field address

>NAME cfa --- nfa code field address to name field address

>VIEW cfa --- vfa code field address to view field address

>OFA cfa --- ofa code field address to optimize field

BODY> pfa --- cfa parameter field address into code field address

LINK> lfa --- cfa link field address to code field address

L>NAME lfa --- nfa link field address to name field address

N>LINK nfa --- lfa name field address to link field

N>CFAPTR nfa --- cfp name field address to code field pointer

NAME> nfa --- cfa name field address to code field address

NFA-COUNT nfa --- adr c count nfa into name count and address

VIEW> vfa --- cfa view field address to code field address

N>OFA nfa --- ofa name field address to optimize field address

Caution: W32F uses n-way threaded word lists. Thus, when searching for a specific name you will only reach 1/n of the words on any specific linked list of word. Below, ‘xt’ refers to Execution Token the address of a word’s code field address. That value is used by Forth’s inner interpreter.

.NAME cfa --- Type a word name from its code field XT address

: .NAME ( xt -- ) \ show name, if can't find name, show address

DUP >NAME DUP NAME> ['] [UNKNOWN] = \ if not found

IF DROP [CHAR] " EMIT ." 0x" 1 H.R [CHAR] " EMIT SPACE

ELSE .ID DROP

THEN ;

: .ID ( nfa -- )

NFA-COUNT TYPE SPACE ;

## Vocabularies

A vocabulary consists of group of Forth word definitions in the dictionary. In ANS Forth vocabularies are called ‘word-sets.’ Vocabularies allow the programmer to create application specific word groups, hide primitives, segregate identically named words, reduce dictionary search time and specify convenient points to FORGET. See Word Searches, below, on the use of vocabularies.

A new vocabulary may be defined in the form:

VOCABULARY USER-VOCAB

11 #VOCABULARY ANOTHER-VOCAB

In the second example, the parameter 11 specifies the vocabulary is to be created with 11 threads (see below), bypassing the default value. The number of threads should be an odd number, preferably prime, to assist in equalizing the number of words per thread.

A vocabulary begins as a pointer in the parameter field of a vocabulary definition pointing to the link field of the most recently defined word of that vocabulary and then through a backward linked list through the prior words. To speed the location of these words, each vocabulary consists of several parallel lists called ‘threads.’ The thread to be searched is determined by hashing on the word length and several letters of its name. The FORTH vocabulary has 157 (more now?) threads to dramatically increase the speed at which its words will be located. When a word is searched in the FORTH vocabulary, its name is analyzed and one of 157 threads is searched. The default number of threads is 7 set by the value #THREADS (a ‘value’ not a variable). By changing this value you may specify the number of threads for a new vocabulary in the range 1..511. W32F allows up to any number of vocabularies to be defined and up to 16 to be available for searching at any given time

The parameter field for a vocabulary is:

Name/use Field Bytes Use

cfa field 0..3 4 points to code for DOVOCABULARY

voc link 4..7 4 forward link to the next vocabulary

num threads 8..11 4 number of threads in this vocabulary

thread 0 12..15 4 points to most recent word in thread

thread 1 etc 16..19 4 continuing for remaining threads. . .

Vocabularies are forward linked from the user variable VOC-LINK via the ‘voc link’ field in each vocabulary’s parameter field.

During interpretation or compiling the dictionary search process is: 1) move to the next vocabulary address in the search order, 2) locate its parameter field, 3) hash on the word length and letters using the vocabulary’s ‘num threads’ value, 4) the hash value selects the thread from the parameter field, and 5) search that thread until finding the word or reaching a zero link.

## Word Searches

The user may easily specify the vocabularies and the order in which they are to be searched during interpretation or compiling. The search order is cleared by the word ONLY. To add the first vocabulary just execute its name. To add another vocabulary to the search order execute ALSO and a vocabulary name. The search order has the capacity of sixteen vocabularies.

ONLY FORTH ALSO MY-VOCAB ALSO NEXT-VOCAB

The search order searched from the most recently added to the first added vocabulary. This example sets NEXT-VOCAB as the first vocabulary searched, then MY-VOCAB and lastly FORTH. The vocabulary into which new words will be added (called the CONTEXT vocabulary) is specified by the word DEFINITIONS. A typical sequence would be:

ONLY FORTH ALSO MY-APPLICATION DEFINITIONS ALSO TOOLS

In this example the search order is TOOLS MY-APPLICATION and FORTH. The context vocabulary for new words is MY-APPLICATION.

Vocabulary names may be compiled into colon definitions to allow easy to switches between specific sequences:

: SETUP-VOCS ONLY FORTH ALSO MY-APPLICATION DEFINITIONS ;

## Forgetting Words In The Dictionary

Removing words from the dictionary and reclaiming memory space is quite involved in Win32F. It involves the memory space split between word headers and system code space, vocabularies and the multithreaded form of individual vocabularies.

W32F uses a ‘smart forget’ used in the form FORGET <word> that will properly unlink: 1) words defined after <word> regardless of the vocabulary in which they are defined, 2) vocabulary names forward linked via their voc-link field and 3) words in all execution chains. Some of the chains trimmed back during a FORGET include: loaded files, deferred words, fonts, and menus. To see the full list execute .CHAINS and refer to FORGET-CHAIN. The key forgetting words are in file NFORGET.F

The following words are used to mark points for forgetting and perform that process.

FENCE A variable holding the memory address below which forgetting is blocked. Used in the form HERE FENCE !.

FORGET <word> Remove <word> and all words defined after <word> and reclaim their memory space.

EMPTY Trim the dictionary back to just after the value in FENCE.

MARKER Create a word the execution of which will forget back to and including that word? MARKER CUT-HERE.

ANEW <word> Forget all words after <word>. If <word> doesn’t yet

exist, place it in the dictionary. If <word> is executed it and all later entries are forgotten without replacement of <word>.

An application, or sections of an application usually begin with a sequence similar to: ANEW DATA-PROJECT. Upon each recompilation of the file the dictionary will be trimmed back to DATA-PROJECT to remove the prior compilation.

## Dictionary Layout and Access

Forth’s dictionary memory is divided into two areas. The SYSTEM portion holds the headers for words definitions. The APPLICATION portion holds the ‘body’ of Forth words, machine code (created by CODE), compiled execution addresses (created by ‘:’ high level definitions) and the parameters of constants, variables and other data structures. Applications deal almost exclusively with the APPLICATION portion of memory.

The variable SDP holds the address of the next available byte in the SYSTEM area; DP holds the address of the next available byte in the APPLICATION space. A number of word pairs allocate memory and compile into the SYSTEM and APPLICATION space:

SYS-HERE APP-HERE return the address of the next unallocated byte.

n SYS-ALLOT n APP-ALLOT allocate n bytes in the memory space

SYS-ALIGN APP-ALIGN move the memory allocation to the next available cell

b SYS-C, b APP-C, compile n into the next available memory byte

w SYS-W, w APP-W, compile w (16 bits) in to the next available memory word.

n SYS-, n APP-, compile n (32 bits) into the next available memory cell.

Because they are used in applications the ‘APP’ prefixed words are repeated in a shortened form: HERE ALLOT ALIGN C, W, and ,

## Advanced Topics

## Header structure

Over the years of Forth evolution a variety of structures have been used for the dictionary. The earliest Forths had a word header with a byte count of the word's name, the first three letters of that name, a pointer link to the preceding dictionary word, a code pointer and the parameter field. The dictionary would grow from low to high memory.

Next came the transition to either variable length or full-length word names in the dictionary with various hashing methods for the links to accelerate dictionary searches. W32F stores each defined word with its full character length. Its words are all stored in UPPER CASE. The user has the choice of compiling new words into upper case (execute CAPS ON) or mixed case (execute CAPS OFF). Forth words may be up to 63 characters long.

Win32Forth splits the dictionary entries with the word's code fields and parameter fields starting at low memory growing upward and the word headers at mid-memory growing upward.

In the header, the name field is adjusted so the vfa, lfa and cfp are aligned to cell boundaries. Immediate words have bit 7 set (hex 80) in the count (nfa). Win32Forth's headers and parameter fields have the following byte ordering and size:

## Structure of Dictionary Fields (as of 2003)

Word name structure in high memory, above C0000h

\ [ link field ] -4 +0 LFA  
\ +- [ cfa ptr field ] +0 4 CFA-PTR  
\ | [ byte flag ] 4 8 BFA  
\ | [ count byte ] 5 9 NFA  
\ | [ the name letters ] 6 10  
\ | [ alignment bytes ] 0 to 3 bytes for name alignment  
\ | [ view field ] n+0 VFA <- head-fields  
\ | [ file field ] n+4 FFA  
\ | [ optimize field ] n+8 OFA

Word code and parameter field in low memory.

\ [ cfa field ] +0 CFA  
\ [ body field ] +4 PFA  
\ -------------------------------------------------------

**Original Dictionary Fields (1995)**

Name Field Bytes

alignment bytes 0..3 filler bytes to align optimize field.

optimize field 4

alignment bytes 0..3 filler bytes to align view field

name 1..n ascii text of a word name

nfa count 1 number of bytes in the preceding word name

vfa view field 4 line number offset into source code file

lfa link field 4 points to the prior word's link field

cfp code field pointer 4 points to code field in user's

dictionary space

**Structure of Data Fields**

Name Field Byte size

cfa code field 4 points to machine code interpreter.

pfa parameter field 4+ or more bytes. May be used for data storage, machine code or a sequence of execution token addresses.

## Use Of Header Fields

Header fields are typically used to locate a word in the dictionary. In a search by FIND, a scan is made through a specific linked list of pointers to match against the input text. The specific linked list (using a chain of link field pointers) is selected by a hashing algorithm. The search routine searches backward, through the earlier defined words stopping at each link. At each link the name length count in the count field is compared to the input word. If they match the full name is compared until a match is obtained or the list is exhausted. These field positions are determined simply by adding or subtracting the proper offset from the link field address (lfa). The search routine FIND returns the target word code field address (execution token) and a TRUE boolean flag or the memory address of the original name text; a returned FALSE flag indicates failure.

A rich set of word are provided to move between these fields. Due to the split dictionary structure and the mixed use of bytes and cells care must be taken to use these words rather than just add or subtract bytes to move between fields.

The most common way to locate a work in the dictionary is by “‘” (apostrophe, pronounced ‘tick’). This sequence ’ DEMO will return the code field address (cfa) of the word DEMO. The following words may be used to locate other fields.

‘ ( --- cfa ) single apostrophe, pronounced ‘tick’

>BODY ( cfa -- pfa ) \ from the cfa get pfa

BODY> ( pfa – cfa) ) \ from pfa get cfa

>NAME ( cfa -- nfa ) \ find the nfa of the provided cfa

NAME> ( nfa -- cfa )

>VIEW ( cfa -- vfa )

VIEW> ( vfa -- cfa )

>OFA ( cfa -- ofa ) \ get the Optimization Field Address

L>NAME ( lfa -- nfa )

LINK> ( lfa -- cfa )

N>LINK ( nfa -- lfa )

N>OFA ( nfa -- ofa )

N>CFAPTR ( nfa -- cfa-pointer )

VLINK>VOC ( voc-link-field -- voc-address )

VOC>VLINK ( voc-address -- voc-link-field )

VOC#THREADS ( voc-address -- #threads )

VCFA>VOC ( vocabulary-cfa -- voc-address )

VOC>VCFA ( voc-address -- vocabulary-cfa )

## CREATE , DOES> AND ;CODE

Not only does Forth contain words to create words (:, CREATE, VARIABLE), you can define a word to create words that define words! This meta-class of words are called ‘user specified defining words.’

In this manner a word using CREATE and DOES> operates at three levels:

**Level 1:** When a word containing CREATE DOES> is compiled, that new word becomes a 'defining word'. One would use CREATE DOES> to create a set of words with a similar execution but with a variety of input parameters. A machine code assembler or data base field specifiers are common uses. A typical definition would appear as:

: OP-CODE CREATE , DOES> @ + W, ;

Care must be taken on the operators storing and reading from DOES> parameter fields. The usual ‘,’ and ‘@’ use 32 bit operands; the more common usage is ‘w,’ and ‘w@’ for 16 bit operands.

**Level 2:** When the Level 1 defining word executes, it uses CREATE to create another word, usually laying down one or more parameters for that new word. The following DOES> defines the run-time action of that new word. Thus, each new word has a unique name, a unique parameter set and shares the common execution following DOES>. Use of our example OP-CODE would be:

HEX 0F0F OP-CODE LDA,

**Level 3**: When the word ‘LDA,’ (defined by the defining word OP-CODE) executes, its parameter field address is automatically placed on the stack and then it executes the sequence compiled after DOES> in LDA,. This allows a variety of parameters to be passed to a common Forth process.

In our example from Level 2, when LDA executes it will fetch the hex value $0F0F from its parameter field add that value to whatever value was on the stack as LDA and then compiles that value as a 16 bit word into the next two 8 bit dictionary bytes. In this example Reg1 provides a number specifying a machine register:

HEX Reg1 LDA,

#### ;CODE

Another defining-defining word construct uses ‘;CODE’. In this case, a defining word uses CREATE to creates a new word and set-up its parameter field. After the high level portion, machine code after ‘;CODE’ defines the machine code the new word will later execute. The ;CODE construct is usually used in the interest of execution speed when multiple parameters are needed.

: 2CONSTANT CREATE , , ;CODE  
push ebx push 2 CELLS [eax] [edi]

Mov ebx, 1 CELLS [eax] [edi]  
next c;

-1 -1 2CONSTANT DOUBLE-FALSE

Later 2CONSTANT will create a double number support word, in this case ‘DOUBLE-FALSE’ followed by two parameters compiled by ‘, ,’ and exits. When the new ‘DOUBLE-FALSE’ runs its execution begins after ‘;CODE’. It extracts the values compiled by ‘, ,’ and places them on the stack. It then returns to the inner interpreter by ‘next’.

## DEFERed words

On occasion the execution code of a word needs to be changed or a place holder words is needed before it has actually be defined. For this purpose a word name in entered in the dictionary. Later its execution is assigned. The syntax is:

DEFER D-WORD \ the placeholder

: A-WORD . . . . ; \ later defined

‘ A-WORD IS D-WORD \ assigning A-WORD’s execution to D-WORD

If a deferred word is executed before being resolved an errror message will occur. However, it can be compiled before being later resolved.

## Use of ;CODE

A specialized Forth construct consists of a defining word (containing CREATE) followed by ;CODE which assigns the machine code following ;CODE as the execution process for the words it later defines. A typical use of a ;CODE would be in the creation of a word to define constants when maximum speed is desired. This structure would appear as:

: CONSTANT CREATE , ;CODE <machine code> NEXT,

$20 CONSTANT BL $01 CONSTANT 1

The same result could be done at high-level using:

: CONSTANT CREATE , >DOES @ ;

In this example CONSTANT creates the word BL, compiles the value $20 into BL’s parameter field and points BL’s code field to <machine code> which exists at the end of CONSTANT. Thus each defined word has its own parameter but shares common <machine code>. Execution for BL begins at the start of <machine code> and must end with NEXT, to continue proper interpretation. In contrast to DOES> (which leaves the parameter field address on the stack) the <machine code> of BL must locate the stored parameter $20 from Forth registers. In this case the parameter $20 is located one cell (4 bytes) past the code field in BL as pointed to by register W.

## CODE Words

If the contents of a code field is the Forth address 4 bytes ahead, then this is a CODE word written in assembler. For a CODE word the parameter field contains machine code ending in the code fragment NEXT or a jump to the code for NEXT located very low in the Forth memory space. Upon execution of a code word, the inner interpreted obtains the contents of the code field and then jumps to that address for the direct execution of machine code. Every Forth Code word is obligated to conclude with the code for NEXT to continue interpretation.

## Recursion

During the definition (compilation) of a new word in the dictionary, the partial definition is blocked from Forth’s FIND word. Thus a prior definition of, say, LINKER can be used in a new definition of LINKER. This allows a prior definition to be expanded in function. The earlier definition will continue to be used if it was referenced in earlier defined words. However the new definition will be compiled and used from that point forward.

Recursion is the construct in which a word (function) calls itself. Forth provides the word RECURSE for this function. Use of RECURSE will compile at its location a call to the most recently defined Forth word (itself). Thus:

: AJAX DUP 1+ DUP 10 < IF RECURSE THEN ; 0 AJAX

will place the numbers 0 through 10 on the stack.

Some Forths block the finding of a partial definition by setting a bit in its name field called ‘smudging.’ Win32Forth accomplishes this by linking the new definition’s link field into the CURRENT vocabulary only upon reaching the definitions’ concluding ‘;’ without error. At that point, in ‘;’, the word REVEAL completes the linking process. The new name is created by “HEADER in which the linking point for the new word is determined and ‘parked’ temporarily in the variable LAST-LINK.

**Return stack**

>R R> R@ DUP>R R>DROP 2>R 2R> 2R@

## Control Flow, Decisions

**Summary**

IF ELSE THEN ENDIF

BEGIN WHILE REPEAT UNTIL AGAIN  
DO LOOP and DO +LOOP ?DO  
EXIT and LEAVE  
[IF] [ELSE] [THEN]  
WITHIN, BETWEEN  
**RECURSE**

**IF, ELSE, THEN Conditional Execution**

Forth’s most basic control structure is used in the form:

<stack value>  
IF <execution if stack value is true, not zero>  
 ELSE <execution if stack value was false, zero>  
 THEN <all execution continues here>

In contrast to other languages the condition test is located before IF. IF accepts and removes a stack value that is either a true (non-zero) or false (zero) value. If true, execution continues ahead until ELSE. If the condition is false the ELSE code executes until THEN. Both cases continue execution after THEN. The ELSE portion is optional.

As for all Forth control structures, IF/ELSE/THEN must be in a compiled (colon) definition. An example:

123 IF .” got non-zero” ELSE .” got zero” THEN

Definite loop, BEGIN AGAIN

Indefinite loop BEGIN UNTIL

Indefinite loop: BEGIN WHILE REPEAT

Definite loop DO LOOP and +LOOP

DO/LOOP defines a looping structure with a progrtess count available. It is formed as:

<limitvalue> <startvalue> DO <actions> LOOP

DO removes from the stack <limitvalue> and <startvalue>. The following code until will execute <limitvalue> minus <startvalue> times. In internal counter will run from <startvalue until <limitvalue> minus 1. That is, the looping will terminate just before reaching the <limitvalue> count. :

: demo cr 10 1 DO i . LOOP ; upon execution will print the integers 1 to 9:

1 2 3 4 5 6 7 8 9

Notice the use of ‘i’. This loop counter is available (only) between DO and LOOP by the Forth word ‘i’. See below.

A caution: If the <limtivalue> is less than the <startvalue>, say: 1 10 DO the looping will take the long way around the number circle, executing for millions of times. The only answer is to crash (close) the system. During development you may want to include in the loop the phrase “key? abort” looping oooops!” to allow recovery by hitting any key.

You must leave the return stack unmodified when reaching LOOP (or using LEAVE or EXIT). This because the return stack is used to count and control looping. That is, you may use >R and R> within a DO/LOOP but balancing an R> for each >R.

## The loop index

Within a DO/LOOP you may obtain the current value of the loop index (counter) by the single letter ‘i’ (index). In the case:

. . . 5 0 DO I . LOOP . . . .

The output would be 0 1 2 3 4. Remember LOOPs end at the value just before the limit value.

DO/LOOPs can be nested in the form:

5 3 DO 5 2 DO 5 1 DO cr i . j . k . LOOP LOOP LOOP

On the first passage you would see 1 2 3. The ‘i’ returns the current (initial) index value of the innermost 5 1 DO loop. The ‘j’ returns the index value of the 5 2 DO loop and ‘k’ returns the index value of the 5 3 DO loop.

## Use of LEAVE

An early exit may be made by the use of LEAVE.

: demo cr 10 2 DO i . i 5 = IF .” we’re done at 5 ” LEAVE THEN ;

Will yield: 2 3 4 5 we’re done

Use of EXIT. I think this is a fatal error???

**CASE for N-way Selection**

The words used for CASE, a one of n selection, are:  
CASE, OF, ENDUF and ENDCASE

CASE selects from one of several execution paths and a default. A CASE structure must be located within a complied (colon) definition.

At each OF, the current two stack values are compared. If equal, execution continues after the OF with the two values dropped. If the two compared values are not equal control, the input test value is maintained and flow continues after the corresponding ENDOF. If the default path is taken the input test value is ultimately removed by ENDCASE.

If the ending default path is taken, note the input stack value will be present until it is finally dropped at ENDCASE. Allow for this value if you are using prior, continuing stack values.

2 ( on the stack at execution time)  
CASE 1 OF .” got a 1” ENDOF  
 2 OF .” got a 2” ENDOF   
 3 OF .” got a 3” ENDOF  
 .” found no match” ENDCASE   
And see ‘got a 2’ as a result. Notice the tested stack value is duplicated at each OF choice and ENDCASE automatically deletes it.

## Execution Chains

Chains are lists of Forth words, each list having a related function, such an numeric input conversion. These words are linked by a sequence of forward address pointers ultimately ending in a value of zero.

All of the chains in W32F originate at the variable CHAIN-LINK. They may be displayed by .CHAINS . The first word shown in this list is the variable serving as the origin for that chain followed by its members. Execution chains are defined in PRIMUTIL.F.

For example, the input number conversion chain begins in the variable NUMBER?-CHAIN. You may list that chain by NUMBER?-CHAIN .CHAIN

In memory, each of the addresses in the chain is followed by the execution address of a Forth word. To execute a chain, the word DO-CHAIN starts at the address of the heading variable. It then sequentially retrieves the next address (location) in the chain, moves one cell (4 bytes) ahead and executes the Forth compilation address at that location. When it reaches a zero value the end of the chain has been reached and the process ends. It is the responsibility of each word in the chain not to disrupt execution of the following words in the chain.

Three words are used to create, add to and execute a chain. The sequence to create a new chain and add a word to that chain is:

NEW-CHAIN DEMO-CHAIN Create a new chain named DEMO-CHAIN.

: X-WORD .” This word will be in the chain” ; An ordinary Forth word.

DEMO-CHAIN CHAIN-ADD X-WORD Add the new word into the chain DEMO-CHAIN.

DEMO-CHAIN DO-CHAIN Execute all the words in the chain DEMO-CHAIN.

Question? Does the FORGET of W32F properly cut back the pointers of chains?

## Error recovery

ABORT  
ABORT”

## CATCH & THROW

[Need more on the use of CATCH and THROW.]

The system numbers assigned to CATCH and THROW are:

-1 EQU THROW\_ABORT

-2 EQU THROW\_ABORTQ

-3 EQU THROW\_STACK\_OVERFLOW

-4 EQU THROW\_STACK\_UNDERFLOW

-13 EQU THROW\_UNDEFINED

: \_MESSAGE ( n -- )

BASE @ >R DECIMAL CR ." Error: "

POCKET COUNT TYPE SPACE DUP THROW\_ABORTQ =

IF DROP MSG @ COUNT TYPE

ELSE ." Error # " .

THEN

?LOADING @

IF BASE @ >R DECIMAL CR ." File: " LOADFILE @ CELL+ COUNT TYPE

." at line: " LOADLINE ? R> BASE ! EDIT-ERROR

THEN

R> BASE ! ;

## Manipulation of memory

ALLOT allocates one byte of memory at HERE.

CMOVE ( from to N --- ) Move N bytes ‘from’ address ‘to’ address starting at the lowest memory address of ‘from’.

CMOVE> ( from to N --- ) Move N bytes ‘from’ address ‘to’ address by starting at highest address in ‘from’ to downward in memory. Used to move overlapping byte sequences.

MOVE ( from to n --- ) Move n bytes ‘from’ address ‘to’ address. Automatically adjust for an overlapping sequence.

FILl (addr n char--- ) from addr to addr+n-1 fill with char

ERASE ( addr n -- ) clear all bits within the byte space

BLANK ( addr n -- ) fill all bytes with ASCII blank character

## Interpretation

Below is the Win32 Forth which interprets each word, whether from the console or from a file. Beginning a loop, the next text in the input stream is parsed ending is a blank or null. While text exists, the word is searched for in the dictionary (FIND). If found, STATE is checked. If true, the found word’s execution address is compiled (COMPILE ,) otherwise it is EXECUTEd. If the text was not found in the dictionary NUMBER converts it into a stack value and optionally compiles it using NUMBER, If none of the prior possibilities are possible NUMBER, will abort giving an error message. The outer QUIT loop handles the input stream and concluding ‘ok’ message.

: \_INTERPRET ( -- )

BEGIN BL WORD DUP C@

WHILE SAVE-SRC FIND ?DUP

IF STATE @ =

IF COMPILE, ELSE EXECUTE ?STACK THEN

ELSE NUMBER NUMBER,

THEN ?UNSAVE-SRC

REPEAT DROP ;

The input stream is parsed, word by word to the string buffer POCKET. The parsing sequence is: BL WORD copies text from the input stream, ignoring leading blanks and ending at the blank after the word’s characters. The variable >IN moves to the end of that text in the input stream buffer. The new word is copied to the string buffer POCKET with the first byte being the string count. WORD returns the address of the count in POCKET. From this address COUNT will conclude with the address of the first character and letter count. A typical use would be BL WORD COUNT TYPE.

## DOS Commands

Many of the original MS-DOS (Microsoft Disk Operating System) commands appear in this Forth. These words do not accept blanks in path names or directory names. They include:

CD <full-path> Change directory to that specified.

CHDIR Synonym for CD

CLS Clear screen (the interactive window)

DIR {text} Display files within the current directory. If the

optional {text} is included, only entries with that

text will be shown. Wildcard characters of \* and ?

may be used. DIR .. moves to the outer directory.

RENAME <f, t1, t2> Within the file name 'f', the text 't1' will be

replaced by text 't2'. Remember, no spaces in file names.

DOS <dos command> Execute dos command in a new window and close it.

## Conditional Execution and Compilation

Several forms are provided to skip over comments and documentation:

( comment ) skip text on the same line

\ comments skip text until the end of the line

\S skip text until the end of file.

(( multiple lines )) skip until )) or reaching the end of file

.( to be printed ) print the comment area during compilation

/\* a comment \*/ comments over multiple lines.

(\* a comment \*) comments over multiple lines.

DOC documentation ENDDOC skip intermediate text

// comment to end of line

-- comment to end of line

[IF] [ELSE] [THEN] conditional execution see below  
\- <word> load line if word IS NOT defined  
\+ <word> load line if word IS defined

Forth’s conditionals (as IF, ELSE, THEN, BEGIN, UNTIL, etc) may only be used within colon definitions. It also has conditional words that are interpreted: [IF] [ELSE] [THEN]. Usually these words are used to select compile time options by selecting source code sections. Consider:

<boolean> [IF] some text [ELSE] more text [THEN]

If the <boolean> is non-zero (true) the text between [IF] and [ELSE} will be processed; otherwise the text between [ELSE] and [THEN] will be processed. Here is an example showing how words may be added only if they are presently undefined in the dictionary:

: undefined ( <name> -- f ) BL WORD FIND NIP 0= ;

undefined BOUNDS [IF] : BOUNDS OVER + SWAP ; [THEN]

Win32Forth has a similar function already included:

\- <word> load line if word IS NOT defined

\+ <word> load line if word IS defined

## Error Codes

While compiling from a file, the number of values on the data stack is checked at the end of each line. An error advisory is given if the number has changed. If attempting to compile a file with a number already on the stack you will also receive this error message. To suppress this report place NOSTACK1 at the end of the line.

Several control words are provided to control error messages.

WARNING OFF **or else** WARNING ON to suppress warnings

NOSTACK and CHECKSTACK for stack warnings during compilation

The Forth ANSI Standard specifies THROW and CATCH for error response to pre-defined messages. This list is abbreviated. See the ANSI Standard for the full list in Section 9.3.5. The Standard cases are:

Error number Message text

-1 no message

-2 message from ABORT"

-4 stack underflow

-13 is undefined

-14 is compilation only

-16 requires a name

-22 control structure mismatch

-38 file not found

-45 floating point stack underflow

-58 unmatched interpreted conditionals

Extended System Error Messages, specific to Win32Forth are:

Error number Message text

-260 is not a DEFER

-262 is not a VALUE

-270 out of memory

-271 memory allocation failed

-272 memory release failed

-280 create-file failed

-281 read-file failed

-282 write-file failed

-290 is interpretation only

-300 locals defined twice

-301 too many locals

-302 missing }

-310 procedure not found

-311 <Windows error message>

-320 stack changed

-330 can't use EXIT in a method

-331 can't use DOES> in a method

-332 method must end with ;M

Warning Messages

Error Number Message text

-4100 is redefined

-4101 is a system word in an application word

-4102 is an application word set to a system word

-4105 has a hash value already recognized by this class."

-4103 stack depth increased

-4104 is a deprecated word

-4106 is an application word whose runtime is in a system word."

Application and Runtime Error Messages

Error Number Message text

9998 Windows exception trapped

## Manipulating the Top of Stack

The data stack values are 32 bit ‘cells.’ WORD-SPLIT will split such a 32 bit cell into its high and low 16 bit components each placed into a 32 bit value with the high 16 bits of each value as zero. The original high 16 bit portion will form the top of stack value. WORD-JOIN will merge the low 16 bits of the top two stack values into one 32 bit stack value. HIGHWORD will extract the high 16 bits to become the low 16 bits of top of stack. LOWWORD will mask out the high 16 bits to retain the low 16 bits.

## User Area

Memory of 4096 bytes is reserved for variables and stack for the user. If the system is extended for multi-user use, then each user has his own dedicated User Area. The User Area location may be determined by CONUSER .

PAD A string buffer of 260 bytes.

HLD A numeric output buffer of 80 bytes.

TIB Terminal input buffer of 260 bytes.

FLOATSTACK Holds the floating point stack, 256 values by 10 bytes each.

## Using Floating Point

The Win32Forth floating point support is based on a floating-point stack values in a 10 byte, 80 bit representation. The arrangement is in the form expected and generated by the computer’s floating-point processor. As an example, try:

123 S>F FDUP SCRATCH F! F.

to see the integer 123 placed on the data stack, converted to 80 bits on the floating point stack, then a duplicate stored in the float variable SCRACH and, finally, displayed in a formatted floating format.

## Float-point Representations

\*\*\*\*\* revise this as a duplicate \*\*\*\*\*

For appropriate uses Win32Forth may represent floating-point numbers: 10 byte ANSI format (normal use), 4 byte 32 bit IEEE format, 8 byte 64 bit IEEE format.

When stored in memory the user may store and recover the full 10 byte value using F! F@ with a variable created by FVARIABLE. In addition, you may store and recover floats in the 4 byte, 32 bit IEEE format using SF! SF@ in a variable created by VARIABLE (the usual integer 32 bit variable). Finally, floats may be stored and recovered in the 8 byte, 64 bit IEEE format using DF! DF@ using a double number variable created by 2VARIABLE.

In all of these manipulations the floating point stack representation is always the 10 byte ANSI format. The 4 byte and 8 byte formats apply when the float is on the data stack. One must be very careful the storage space will accommodate the stored values lest nearby memory be overwritten.

## Summary

Usual floats: F!, F@, FVARIABLE, FDUP (on the floating point stack)  
4 bytes: SF!. SF@. VARIABLE, DUP (on the data stack)  
8 bytes: DF!, DF@, 2VARIABLE, 2DUP (on the data stack)

In the following words ‘n’ represents a 32 bit data stack number, ‘d’ a double data stack number, ‘addr’ a data stack address, ‘r’ a (real) number on the floating point stack. Note there are two words (FS>DS, SFS>DS) which will produce an IEEE standard floating point number of 64 or 32 bits on the data stack from a floating point stack number. This would be useful for writing files and passing to a DLL in the IEEE standard format.

F. Display a floating point number r.

FE. Display r in engineering notation.

E.

G. Display r in a generalized format.

FS. Display r in scientific notation.

.FDEPTH Display the number of values on the floating point stack.

F.S Display the contents of the floating point stack.

S>F Convert a stack number n into floating point r.

D>F Convert a stack double number d into floating point r.

F>S Convert a floating point r into a stack number n.

F>D Convert a floating point r into a stack double number d.

FS>DS Move floating point r to the data stack as 64 bit float IEEE.

SFS>DS Move floating point r to the data stack as 32 bit float IEEE.

F# Accept the following text at a floating point number.

SET-PRECISION Set the number of digits after the decimal for floats output.

FCONSTANT Create a floating point constant from value r on float stack.

VARIABLE Create a variable to hold a floating point value.

F@ Fetch the floating point value at addr.

SF@ Fetch an IEEE 32 bit floating point value.

DF@ Fetch an IEEE 64 bit floating point value.

F! Store the floating point number r at address addr.

SF! Store the float in the IEEE 32 bit format at addr.

DF! Store the float in the IEEE 64 bit format at addr.

FDEPTH . Display the number of values on the floating point stack.

FPI Place the value of Pi on the floating point stack.

FL2T Place log base 2 of 10 on the floating point stack.

FL2E Place log base 2 of e on the floating point stack.

FLOG2 Plase log base 10 of 2 on the floating point stack.

FLN2 Place the natural log ln2 on the floating point stack.

Note: the following words are in the HIDDEN vocabulary.

F\*\* Raise r1 to the power of r2 producing r3.

F\*\*N Raise the value on the floating point stack to the power

of the integer value on the data stack.

Note: the following words are in the HIDDEN vocabulary

FDUMP Display the contents of the Pentium floating point processor.

F.X Display the floating point stack in hex.

## Floating Point Internals and Details

FLOATSTACK is the base of the floating point stack, had allocated 2560 bytes and thus may hold 256 floating point numbers. User variable FLOATSP holds the relative offset top of the floating point stack. B/FLOAT specifies the size of a floating point value, set to 10 bytes, 80 bits, a direct representation of the values into and out of the Pentium floating point processor. FSTACK is a constant holding the location of the floating stack relative to the start of the user area.

## File Access Details

As each file is loaded a simple dictionary entry is made consisting of a backward pointer to the prior fund locator and the file name as a counted string. The variable LOADFILE points to the most recently created file. This sequence is used by VIEW to display the word’s source code definition.

## Assembly Code Level Internal Details

Win23Forth includes an assembler to facilitate programming in machine code. We might choose to use machine code for program speed (most common) or brevity. This material is from *A Beginner’s Guide to Forth* by Dr. Julian Noble.

Win32Forth’s assembler offers the option of either the classical prefix notation (opcode, destination, origin) or postfix notation (destination, origin, opcode). The system itself was complied using the prefix style so as to be comfortable for experienced assembly language programmers.

It is important to realize assembly language conventions differ from one Forth version to another. Moreover the instruction set will be particular to a given target computer. That is, there is no such thing as a generic assembler in any programming environment, much less for Forth. Hence everything we do here will be specific to Win32Forth running on a Pentium-class machine.

Conventions followed: register eax serves an an accumulator for arithmetic operations; register edx holds the USER area address; register ebx holds the top of stack 32 bit value; xx is the stack pointer; register ecx is a scratch register used for temporary storage.

Suppose a program uses the sequence \* + many times. Obviously good factoring practice would dictate that this sequence be given its own name and defined as a new subroutine (word). So we might define

: \*+ \* + ;

and substitute it for the sequence \* + throughout the program. But suppose we discover that this short sequence is the bottleneck in our program's running time, so that speeding it up will greatly increase speed of execution. (Of course it isn’t likely in our simple example.) To translate this to machine code we first look at the machine code for \* and + separately. These are primitive words and almost certainly will be CODE definitions in any reasonable Forth.

Thus we need to disassemble these words. Win32Forth has a built-in disassembler. If we SEE a CODE definition it will return the actual byte-codes as well as the names of the instructions in the Win32Forth assembler. Let us try this out:

see +

+ IS CODE

4017AC 58 pop eax

4017AD 03D8 add ebx, eax

see \*

\* IS CODE

401B9C 8BCA mov ecx, edx

401B9E 58 pop eax

401B9F F7E3 mul ebx

401BA1 8BD8 mov ebx, eax

401BA3 8BD1 mov edx, ecx

To understand these sequences we must bear in mind Win32Forth keeps TOS in a 32-bit ‘ebx’ register. We must also know that Win32Forth uses the ‘edx’ register for other tasks. So if our program is going to modify the edx register, its previous contents have to be saved and later restored. Since addition of eax to ebx does not affect edx, the CODE for + doesn't need to protect edx; however, when two 32-bit numbers are multiplied, the result consists of 64 bits with this product ending in registers eax (bits 0 through 31) and edx (bits 32-63).

This is the reason for saving edx into the unused ecx register, and then restoring it afterward.

We note the Win32Forth assembler follows the Intel convention:

mov destination, source

Therefore: add ebx, eax

adds the contents of register eax to ebx, leaving the result in ebx (which conveniently is the top of the data stack). There are similar sequences

mov ecx, edx and

mov ebx, eax

We should also ask why the integer multiplication instruction

mul ebx

has only one operand? The answer is that the register ‘eax’ acts as an accumulator, as initially it contains the multiplier and then it and register ‘edx’ contain the final product, as noted above. It is only necessary to specify where the multiplicand originates (it could be another register or a cell in memory).

To define the word \*+ in assembler we would type in:

CODE \*+ ( a b c -- b\*c+a) \ stack: before – after  
mov ecx, edx \ protect edx because mul alters it  
pop eax \ get item b; item c (TOS) is already in ebx  
mul ebx \ integer multiply-- c\*b -> eax (accumulator)  
pop ebx \ get item a  
add ebx, eax \ add c\*b to a -- result in ebx (TOS) –done  
mov edx, ecx \ restore edx  
next, \ return to the calling Forth word  
END-CODE

Note that the Forth assembler uses the backslash ‘\’ to denote comments follow. This because the common ‘;’ is a Forth word with another use.

The word END-CODE has an obvious meaning, but what about ‘next,’ (the comma is part of the name and is significant!). The word ‘next,’ complied in place a macro that returns control to the Forth word calling this code sequence. It is analogous to CALL and RETURN in classical assembler use.

## Bibliography

Dr. Julian V. Noble, *A Beginners Guide to Forth*, 2001, The best overview of Win32Forth so far. Published on-line at:

<http://galileo.phys.virginia.edu/classes/551.jvn.fall01/primer.htm#code>

James D. Terry, *Library of Forth Routines and Utilities*, Plume 1986, 374 pages, ISBN-10: 0452258413, ISBN-13: 978-0452258419. Out of print but available on Amazon from used book sellers. Has an extensive section on Forth assembly language and use of the 808x floating point processor. It appears to have formed the bases of the floating point for Win32Forth. If that was the case, the current version has been expanded into a dedicated floating point stack and almost exclusive use of floats as 10 byte, 80 bit values.

William Ragsdale, *The fig-FORTH Installation Manual*, 1980, Forth Interest Group, out of print. While written for an 8 bit microprocessor with a 16 bit data stack this model forms the basis of many later Forth implementations on 16 and 32 bit computers.. Clearly documents Forth’s internals including the inner (machine code) interpreter and the outer (text/compilation) interpreter. Available on line at: <http://home.hccnet.nl/a.w.m.van.der.horst/figdoc.zip>

## On-line Resources

A very rich page of Forth links:

<http://pages.cs.wisc.edu/~bolo/forth/>

The Taiwan Forth Group has fully documented the words of Win32Forth [Try Chrome for direct translation]:

<http://www.figtaiwan.org/Win32Forth61110/doc/>

A Beginners Guide to Forth by Dr. Julian V. Noble

<http://galileo.phys.virginia.edu/classes/551.jvn.fall01/primer.htm#code>

## About The Author

Bill Ragsdale was one of the five founders of the Forth Interest Group in 1978. He was the first FIG president and guided it for six years. During that time FIG grew to over 3,000 members in 45 chapters, world-wide.

Bill created the fig-FORTH Implementation Model and led the Implementation Workshop at which the model was translated for twelve processor types including Intel 8080, DEC PDP-11, Motorola 6800 and the MOS Technology 6502. Bill later became part of the Forth 79, Forth 83 and ANSI Standards Teams.

He is the originator of ONLY and ALSO first presented at the 1982 Asilomar FORML Conference. See <http://forth.sourceforge.net/standard/fst83/fst83-c.htm>

Until his retirement in 1995 Bill was the president of Dorado System Corp. which used Forth in all of its security and communications products.

Bill may be reached at [bill@billragsdale.cc](mailto:bill@billragsdale.cc) or 530-867-6241 in Woodland, CA.

## Open Items and Questions

## Note on setting fonts

[Found on-line.] I have used Zimmer's 4.2 version, I was setting the console font by set-font, I tried this, but it doesn't accept my section, while this word is deprecated. How can I set the font to a more readable (bigger) one?

I don't know which version you are using now, but try this:

Font cFont  
16 Height: cFont  
8 Width: cFont  
s" Courier New" SetFaceName: cFont  
FW\_NORMAL Weight: cFont  
Create: cFont  
Handle: cFont SetConsoleFont  
zHandle: cFont