Using Win32Forth

By Bill Ragsdale

Table of Contents

***** THIS IS A WORK IN PROCESS ***** updated 05/28/2022

******* EXPECT IRREGULARITIES ****** You will see notes to myself and locations for future work.

Chapter 1 System Information and History

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 European programmers.

Introduction

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 and will run on Windows 10. I recommend the latest release.

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 may be downloaded from http://www.github.com/billragsdale.

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 started to use early Win32Forth and 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. This is more of a reference document than 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 this 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. To refer to the ANS Standard, see Help below.

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’.

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. These books are 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.

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. (Plus 'locals' and vocabulary stacks.)
- 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.
- Extensive testing and debug support with 's.', 'see', 'view', 'debug' and 'abort"'.
- Trig and transcendentals support.
- N-way hashed vocabularies for search speed. ONLY & ALSO
- Object oriented programming.
- Windows constants and linkage. Some 20,000 of them!

Chapter 2 Getting Started

How To Install

Installing and First Use, editing, two window structure.

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 (?), updated 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 Win32F IDE' 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`** and then Esc 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 your way to your Win32Forth adventure.

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

Help

A rich selection of support is included, but is a bit obscure to find. Hit F1, then Help, then Help and see this list (plus more) :

Help on Help
Overview
Getting Started
Win32 Reference — It is essential you become familiar with this material.

Win32F IDE	Use of the Integrated Development Environment
WinEdit	The original Forth editor.
ANS Standard	The Forth Standard of about 1993 on which W32F is based.

It is ESSENTIAL you open the Help system then open and expand each choice. You will find resources beyond your imagination.

What is Included

The following words are executed at W32F startup to show your current Forth environment. You may try then now. The values below were for my currently running system.

`.version`	Version: 6.15.05 Build: 2
`.cversion`	Compiled: Saturday, August 29 2020, 7:08PM
`.platform`	Platform: Windows 10 Home
`.words`	2,814 Words in Application dictionary, 2,634 Words in System dictionary, 5,448 Words total in dictionaries, 20,312 Windows Constants available.

Chapter 3 Key Components

Key Components: Follows ANS-conventional, stack, dictionary, words, stack notation, two interpreters, RPN math, memory, editing, number formats, control structures, file access

The Forth Dynamic

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. W32F also supports floating point numbers.

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 (called ‘execution tokens’) 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.

Editors

W32F comes with two editors: WinEd, its original editor and Win32F IDE, adding support for project management and the object-oriented program module. Both support cut and paste (cont-C, cont-V, cont-X) and the mouse scroll wheel. Win32F IDE adds visible line numbers plus select and drag text. Both have Help links to documentation.

After editing a file just hit F12 to compile and execute the program. You will find this process very convenient and F12 will become you best friend.

My installation has this quirk: If editing and executing from WinEdit and an error occurs the Forth program error report process opens Win32F IDE and highlights the offending line of text. You now have two open copies of the same program and the risk of them being edited independently. This pretty well forces the use the IDE version.

My Win IDE installation also has the problem printing of a text file uses microscopic text and is unusable. To print I copy to Notepad++. WinEd appears to print well but with more limited control choices.

The Word file for this book uses TimesNewRoman for text. It uses **Fixedsys Excelsior 3.01** for Forth source code.

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 F12 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.

Some Words To Try

WORDS	Display the words in the Current (most recent) vocabulary. Use ‘esc” to exit. ‘Space’ to pause.
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.
VIEW <word>	Open the file containing that word and highlight it.
SEE <word>	Disassemble and display the following word.

Chapter 4 Your First Usage stacks, manip, defining

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 later about two more stacks. Our discussion will focus on the Parameter Stack, again, 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-bit, twos-complement integer. In unsigned hexadecimal, 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 64 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 **ggg 'enter'.**

As a bonus, at the bottom of the Forth output window you will see a brief report on the condition of the user console.

```
Base: decimal   Stack: [3] 2 4 6   Floating point stack: empty.
```

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.
RESET-STACKS	(---) Empty the data stack. The return stack is NOT changes.

The Two Stacks

A stack is a sequential memory allocation accessed from one end by adding and deleting items. Think PUT and POP although these terms are not used in W32F. W32 stacks are:

Data Stack – the core Forth activity for holding numbers, characters, address values, etc.

Return Stack --- Controls the nesting as one word calls another. Executing a word puts the address of its calling word on the return stack. Exiting a word returns control, via the address on the return stack, to the calling word. The return stack is also use WITHIN a single definition to hold temporary values, using >R, R@ and R> and loop control parameters.

Floating Point Stack --- This stack holds 8-byte floating point numbers.

Search Order Stack --- When I invented the search order using ONLY and ALSO, I did not envision it as a stack, but that usage has evolved by common use. Originally there was no way to remove a vocabulary from the starch order stack. My intent was that you would clear and re-declare the search order. And I suggested that search order could be compiled such as:

```
: FOR-EDITING   only forth also applications also editor ;
```

The word ORDER displays the current search order. In response to user demand the word PREVIOUS removes the most recently added word in the search order. The following sequence would add and remove Editor from the search order.

```
Only forth also application also editor    [ some work ]  previous
```

And the search order is now `forth application`

Word Types

Forth offers a rich variety of words: Defining words, data words (i.e. constants, variable and more), control flow words (IF, DO-LOOP, BEGIN, etc.), math operators (+, -, *, /, etc), logical operators (and, or, xor, etc.), equivalent floating point words, stack manipulation (over, swap, >r), string support, file access, programming support words (file, include, words, dump, etc.), a machine code assembler and object oriented support.

Defining Forth Words

*** Add section on colon definitions *** and ;code.

The defining words: CODE ':' and CREATE add execution code to the dictionary with a given name. Later interpretation (activation or execution) of that name begins execution of the associated compiled code. That code may be actual machine code or a sequence called 'execution tokens'. Machine tokens ultimately link to and activate machine code.

The words CONSTANT, VARIABLE, and VALUE and words created by CONSTANT hold data in a variety of formats.

A similar set of words for floating point numbers are: FCONSTANT, FVARIABLE and FVALUE.

Yet another set of defining words are supplied to hold 64 bit signed integers (double numbers): 2CONSTANT, 2VARIABLE and 2VALUE.

and creator-creator words (CREATE+DOES>, and CODE-;CODE).determining their function. They include code words, colon definitions, data types (CONSTANT, VARIABLE, VALUE),

***

*** move the following to other categories ***

Compound words: CREATE + DOES>, CREATE + ;CODE, and IS + DEFER
Assembler words: ADIW, IF,
Label optional, may be used to label un-named assembly code sequences
***

Use of CONSTANT, VARIABLE, VALUE and :

Constants will add their numerical value to the data stack. A Variable gives the address of a stored value that may recalled and re-written. A Value will yield its stored value but can be re-written; thus, it has characteristics of a Constant and a Variable. Words created by Create have a selectable storage area suitable to hold numbers, text strings, arrays and more (as programmed).

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. **2VARIABLE <var-name>** creates a double number variable, reserving 2 cells, 8 bytes. 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.

Value

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.

2VARIABLE and 2CONSTANT

2**VARIABLE <var-name>** creates a double-number constant <var-name> reserving 2 cells, 8 bytes. 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 2variable:
2**VARIABLE demo 12345. demo 2! 'enter'**

<2number> 2CONSTANT <con-name> will create a double number, named constant. Its execution will place a 64 bit, 2 cell, 8 byte value on the stack.

Use: **12345. 2VARIABLE <2name>** Execute **2name D. and see 12345**

Math Operations

Math operations occur on the data stack. One or two values are accepted and one or two values are returned. 'n' values are signed, 32 bit values, 'un' are unsigned 32 bit values, 'd' are signed 64 bit values, 'ud' are unsigned 64 bit values.

+	(n1 n2 -- n3) Add n1 to n2, return sum n3.
-	(n1 n2 -- n3) Subtract n2 from n1, returning the difference n3.
*	(n1 n2 -- n3) Multiply n1 by n2, returning the 32 bit product n3.
/	(n1 n2 – n3) Divide n1 by n2 returning n3 the 32 bit quotient n3.
M*	(n1 n2 -- d1) Multiply the signed n1 by n2, returning the double result d1.
UM*	(u1 u2 -- ud1) Multiply unsigned u1 by unsigned u2 returning the double ud1.
/MOD	(n1 n2 -- rem quot) For 32 bit signed values, divide n1 by n2 with remainder & quotient.
MOD	(n1 n2 -- rem) For 32 bit integers divide n1 by n2 returning the remainder.
UM/MOD	(ud1 u1 -- rem quot) Divide unsigned double ud1 by the unsigned number u1 returning the 32 bit signed remainder and quotient.
*/	(n1 n2 n3 -- quotient) Divide unsigned double ud1 by the unsigned number u1 returning the signed 32 bit quotient.
*/MOD	(n1 n2 n3 -- remainder quotient) Integer single multiply and divide: Return the signed 32 bit remainder and quotient of [n1*n2]/n3. The intermediate result n1*n2 is a double, so there is no overflow.
FM/MOD	(d n -- rem quot) For signed values, divide 64 bit d1 by the 32 bit n1, giving the 32bit floored quotient n3 and the remainder n2.
SM/REM	(d n -- rem quot) For signed values, divide 64 bit d1 by 32 bit n1, giving the 32 bit signed symmetric quotient and the remainder.
NEGATE	(n1 -- n2) Negate n1, returning the 2's complement n2.
ABS	(n -- |n|) Return the absolute value of n1 as n2.
1+	(n1 -- n2) Add one to n1.
1-	(n1 -- n2) Subtract one from n1.
2*	(n1 -- n2) Multiply n1 by two.
2/	(n1 -- n2) Signed divide n1 by two.
U2/	(n1 -- n2) Usigned divide n1 by two.
D2*	(d1 -- d2) Multiply the double number d1 by two.
D2/	(d1 -- d2) Divide the double number d1 by two.
WORD-SPLIT	(u1 -- low high) Split the unsigned 32 bit u1 into its high and low 16 bit

The following comprise the ANI Standard Double Number word set.

D+	(d1 d2 -- d3) \ add 2 doubles - no overflow check
D-	(d1 d2 -- d3) \ substract 2 doubles
DNEGATE	(d1 -- d2) \ negate d1, returning 2's complement d2
DABS	(d1 -- d2) \ return the absolute value of d1 as d2
S>D	(n1 -- d1) \ convert single signed single n1 to a signed double d1.
D>S	(d -- s) \ convert double to single
D=	(d1 d2 -- f1) \ f1=true if double d1 is equal to double d2
D0<	(d1 -- f1) \ Signed compare d1 double number with zero.
D0=	(d -- f) \ double compare to 0
D<	(d1 d2 -- f) \ Signed compare two double numbers.
D>	(d1 d2 -- f) \ Signed compare two double numbers.
D<>	(d1 d2 -- d) \ Signed compare two double numbers.
DMIN	(d1 d2 -- d3) \ Replace with the smaller of the two (signed).
DMAX	(d1 d2 -- d3) \ Replace with the larger of the two (signed).
2CONSTANT	(n1 n2 --) \ create a double constant
2VARIABLE	("name" --) \ create a double variable
2LITERAL	
2VARIABLE	
D.	(d --) \ display as signed double.
D.R	(d w --) \ display as signed double right justified in w wide field
D>S	yes
DNEGATE	
M*/	yes
M+	

Triple Number Word Set

Extended precision refers to triple sized integers comprising three 32 bit cells, for 95 bit plus sign. ‘t’ is a signed triple (96 bit integer); ‘ut’ is an unsigned triple integer;

This extended precision math word set was developed by Robert Smith who also did the major work on the W32F floating point.

TNEGATE	(t1lo t1mid t1hi -- t2lo t2mid t2hi)
UT*	(ulo uhi u -- utlo utmid uthi)
MT*	(lo hi n -- tlo tmid thi)
(UT/)	(utlo utmid uthi n -- d1 n)
UT/	(utlo utmid uthi n -- d1)
M*/	(d1 n1 +n2 -- d2)

M+ (d1 n -- d2)

8.6.2 Double-Number extension words

2ROT

2VALUE

DU<WORD-JOIN (low high -- n1) Join the high & low 16 bit quantities into a single 32 bit n1.

Examples of words using strings.

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)
```

Control Flow, Decisions

Conditionals Summary

'Conditionals' refer to the Forth Words that either make decisions or control looping. Forth has a rich selection of these words and structures for the control of program flow. Briefly, they are: **IF, ELSE, THEN** for conditional flow; **BEGIN WHILE REPEAT UNTIL AGAIN** for indefinite looping; **DO ?DO, LOOP** and **+LOOP** for definite looping, and **CASE** for an n-way flow selection.

When included in a loop **START/STOP** will pause and continue execution upon 'space-bar' action. **EXIT** and **LEAVE** may be used in their corresponding loop structures. **[IF] [ELSE] [THEN]** allow text interpretation to be selected based on an *input condition. This form is usually used to selection major options in the selection of source code. Finally, **RECURSE** allows recursive code.

IF, ELSE, THEN Conditional Execution

Forth's convention is any non-zero value has the logical condition TRUE while a zero value has the logical condition FALSE. Forth's basic control structures use logical values in the form:

```
<value> IF <code if true> THEN <all execution continues here>
<value> IF <code for true value> ELSE <code for false value> THEN
<all execution continues here>
```

In contrast to other languages the condition test occurs before IF. IF accepts and removes the logical stack value that is either a true (non-zero) or false (zero) value. If true, execution continues until reaching ELSE or THEN. If ELSE is present, execution jump to after THEN. If the logical input at IF was false (zero) execution jumps to ELSE and execution continues there until reaching THEN. In either case execution resumes after THEN. The ELSE portion is optional.

IF	(flag ---) If flag is true (non-zero) continue execution. Upon reaching the corresponding **ELSE** skip to the corresponding **THEN**. If false (zero) skip ahead to the corresponding **ELSE** or **THEN**.
-IF	(flag --- flag) Duplicate the flag and perform **IF**. This form is equivalent to **DUP IF . . .**
ELSE	(---) Continue execution until the corresponding **THEN.**
THEN	(---) Marks the conclusion of a conditional structure.

As for all Forth control structures, **if/else/then** must be in a compiled (colon) definition. An example:

```
: demo  123 IF ." got non-zero" ELSE ." got zero" THEN ;
```

Upon executing demo, see 'got non-zero'

Control structures may be nested:

```
 if . . . if . . . else . . . then   else . . . then
 begin . . . if . . . else . . . if . . . then . . . then . . . until
```

Indefinite Loops

In the examples below all three structures produce the same user interaction. They show how the various control points can be used.

BEGIN-AGAIN

The structure BEGIN <code> AGAIN define a looping structure that will repeat <code> indefinitely. The outer interpreter of Forth is along the line of (simplified):

```
: QUIT BEGIN  query interpret dup? if number then AGAIN ;
```

The only exit from this sequence is an error. In that case the stacks are cleared QUIT is re-entered.

In usual programming an exit is made by EXIT which will exit the BEGIN-AGAIN loop.

```
: demo   cr ." Hello"  BEGIN cr ." type x to exit "
          key ASCII x = if cr cr ." bye" exit then  AGAIN ;
```

BEGIN-UNTIL

The structure is BEGIN <code> <value> UNTIL. If a true <value> is on the stack at UNTIL the exit is made, continuing execution ahead. If <value> is FALSE execution returns to BEGIN.

```
: demo   cr ." Hello"  BEGIN cr ." type x to exit "
                key ASCII x =  UNTIL  cr cr ." bye"  ;
```

BEGIN-WHILE-REPEAT

The structure is BEGIN <code> <value> WHILE <code> REPEAT. If a true <value> is on the stack at WHILE execution continues ahead to REPEAT and then returns to BEGIN. If <value> is FALSE execution jumps to after REPEAT.

Indefinite loop: BEGIN <code> <value> WHILE <code if value false> REPEAT

```
: demo  cr ." Hello"  cr ." type x to exit "
 BEGIN key  ASCII x = 0= WHILE cr ." type x to exit "  REPEAT  cr cr ." bye" ;
```

Definite loop DO LOOP and +LOOP

DO/LOOP defines a looping structure with a progress 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. That is, the looping will terminate just before reaching the <limitvalue>. An in internal counter will run from <startvalue> until <limitvalue> minus 1.

```
: 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.

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

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.

START/STOP, when included in a loop, will pause and continue execution upon 'space-bar' action. This is useful for testing. **ESC** will **ABORT** and return to the console.

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.

Nested Loops

Discuss loop indices i, j and k.

```
: demo  4 0 do  3  0 do  j . i .  ascii , emit loop loop ;  and see:
0 0 , 0 1 , 0 2 , 1 0 , 1 1 , 1 2 , 2 0 , 2 1, 2 3 , 3 0 , 3 1 , 3 2
```

+LOOP

sfasf

```
: demo  13 0 DO  i  .  3 +LOOP ;  see 0 3 6 9 12
```

Declining Loop

sfs

```
: demo   0 6 DO  i  .  -1 +LOOP ;  6 5 4 3 2 1 0
```

Use of LEAVE

An early exit from a DO-LOOP may be made by the use of **LEAVE**.

```
: demo  cr
 10 2  DO  i  .   i  5  =  IF ." we're done at 5 "  LEAVE THEN LOOP ;
```

Will yield: 2 3 4 5 and we're done at 5

Note: LEAVE will exit one loop. If you have nested loops and wish to fully exit, use a form similar to:

```
. . . <value> IF LEAVE THEN LOOP <value> IF LEAVE THEN LOOP . . .
```

CASE for N-way Selection

The words used for CASE, a one of n selection, are:

```
CASE,  OF,  ENDOF 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, the input test value is

maintained and flow continues after the corresponding **ENDOF**. If none of the **OF** tests are taken the (last) default path is taken.

If the ending default path is taken, note the input stack value will be present until it is finally explicitly dropped before **ENDCASE**.

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" DROP ENDCASE
```

And see **'got a 2'** as a result..

Stack notation in definitions

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 stack notation in parenthesis give a short version of hte word's operation. Values n1 and n2 are input stack parameters, the three dashes represent the execution of <word-name> and n3 remains on the stack after execution. Thus the notation for ! (store) would be:

! **(n1 addr1 ---)** In this case ! (store) stores n1 at memory address addr1 and leaves nothing on the stack. For @ (fetch) the notation would be:

@ **(addr1 --- n1)** The action for @ (fetch) is: memory address 'addr1' is on the stack as @ is executed. The contents of memory address addr1 remains on the stack after execution.

? **(addr1 ---)** The action for ? (query) is: memory address 'addr1' is on the stack as ? is executed. The contents of memory address addr1 is fetched and displayed. Finally, '?' leaves no values on the stack.

Often a one-line description will follow. This format usually appears in source code for each word.

2DUP **(n1 n2 --- d1 n2 n1 n2)** **** Duplicate the top two 32 bit values.

Return Stack Operations

>r (n1 --) (R: -- n1) Move the top data stack value to the return stack.

R> (-- n1) (R: n1 -- Move the most accessible return stack value to the data stack.

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

R@ (-- n1) (R: n1 -- n1) \ get a copy of the top of the return stack

2>r (n1 n2 ---) Copy the top two stack values to the return stack..

2>R (n1 n2 --) (R: -- n1 n2) \ push two items onto the return stack

2r> (--- n1 n2) Recover two values from the return stack.

2R> (-- n1 n2) (R: n1 n2 --) \ pop two items off the return stack

DUP>R (n1 -- n1) (R: -- n1) \ push a copy of n1 onto the return stack

R>DROP (--) (R: n1 --) \ discard one item off of the return stack

RDROP see **R>DROP** a synonym

2R@ (-- n1 n2) \ get a copy of the top two items on the return stack

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 care must be taken using DO-LOOP as it also stores control parameters on the Return Stack. The return stack level must be the same when entering at DO as exiting at LOOP.

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> adds (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	@	!	,
Double cell	64	2@	2!	
Float, 8 bytes	64	F@	F!	F,
Fetch & sign extend	16>32	SW@		
Increment			+!, F+!	
			C+! W+!	
Display cell contents			?	

Memory Operators Details

Refer to the table above for the memory size used by each operator. In all cases the stack value is 32 bits, 4 bytes and will be padded with leading zeros as needed.

@ **(addr --- n1)** Applies to **c@, w@, @, 2@ and SW@** Fetch to the stack the value n1 at memory addr. Used in the form: **VarName @**.

C@

W@

2@ (addr --- d1)

F@ **(addr--- fs: f1)** From the memory location specified by the address on the data stack, fetch floating point number onto the float stack.

?	**(addr ---)** Fetch the 32 bit value at memory addr and displace on the console. Used in the form: **VarName ?**
!	**(n1 addr ---)** Applies to **c!, w!, !, and 2!.** Store the stack value at addr, observing the specific storage size.
C!	
W!	
F!	**(addr fs: n1 ---)** Used in the form: **1234 VarName !**
+!	**(n1 addr ---)** Increment the 32 bit value at addr by the 32 bit stack value n1. Used in the form **1234 VarName +!**
C+!	**(n1 addr ---)** Increment the 8 bit byte value at addr by the low 8 bits of stack value n1. Used in the form: **FFh VarName +!**
W+!	**(n1 addr ---)** Increment the 16 bit word value at addr by the low 16 bits of the stack value n1. Used in the form **FFFFh VarName +!**
F+!	
,	
C,	**(n1 ---)** Applies to **c, w, and ,** sfasfsd
W,	
F,	
to	**(n1 <value) ---)** sfsadf
+to	**(n1 <value> ---)** sdfsd
f->	(n1 "name" --) Store a value into a float variable by name, used in a colon definition.
f+>	(n1 "name" --) Increment the value of a float variable by name. Used in a colon definition.
f#->	(n1 "name" --) Store a value into a float variable by name. Used in a colon definition.
f#+>	(n1 "name" --) Increment the value of a float variable by name. Used in a colon definition.

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 a 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.

64 bit S tack 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 utilize the CPU's floating-point processor, so they execute at full computer speed.

FDEPTH Return the quantity of floats on the floating point stack.

S>F Single (32 bit) integer to a float.

F>S A float to a single (32 bit) integer

D>F A double (64 bit) integer to a float.

F>D A float to a (64 bit) integer.

FDUP FDROP FSWAP FOVER FROT FPICK FNIP Floating point stack manipulations equivalent to their integer counterparts.

FMAX FMIN FABS FLOOR FCEIL FTRUNC FROUND Floating point operations equivalent to their integer counterparts.

FARIABLE FCONSTANT FVALUE FTO Floating point entities equivalent to their integer counterparts.

F+ F- F* F/ (f1 f2 --- f3) Floating point math operations.

F= F< F> F<= F>= F0= F0> F0< Floating point logical tests.

F@ F! F+! F, Floating point memory operators equivalent to the integer counterparts..

SF@ DR@ SF! xxx

F0.0 F1.0 F2.0 Fpi Floating point constants.

F# **Floating point input from the console in the form of: F# ##e, ##.#e, and ##.##e0. IF in a definition the following value will become a floating literal.**

F. FE. FS. F.S Floating point output. See *Display and Printing Format*.

FNEGATE 1/F F2* F2/

2F>D FS>DS SFS>DS Floating point conversions.

>FLOAT PRECISION SET-PRECISION Floating point controls.

F2DROP D2SWAP F2NIP

FBIG FEPS FSMALL Floating point constants

FSIN FCOS FTAN FSINCOS Floating point trig functions.

FASIN FACOS FATAN FATAN2 FASINH FACOSH FATANH

F^2 FSQRT FLN FLNPI FLOG FALOG FEXP Exponential operations

```
FEXPMI  F** FL2T  FL2E  FLOG2  FLN2
```

```
: f->        ( n1 "name" -- )       \ store a value into a float
          ' ?float  1 cells+ cfa-comp, ; immediate

: f+>        ( n1 "name" -- )       \ increment the value of a float
          ' ?float  2 cells+ cfa-comp, ; immediate

: f#->       ( n1 "name" -- )       \ store a value into a float
          ' ?#float  1 cells+ cfa-comp, ; immediate

: f#+>       ( n1 "name" -- )       \ increment the value of a float
          ' ?#float  2 cells+ cfa-comp, ; immediate
```

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 signed double integer is:

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

The most negative signed 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 8 byte, 64 bit ANSI Floating Point Standard. The numeric range is 2.23 x 10^{-380} to 1.79 x 10^{308} (?) with a safe maximum of 16 (see below) decimal 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.
H.2	Display two digit HEX number
H.4	Display four digit HEX number
H.8	Display eight digit HEX number
U.	Display the top of stack as an unsigned integer.
H.	Display the top of stack in hex.
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.
(F.)	(addr -- ; fs: r --) Format float 'r' as a fixed point, counted string in the buffer whose address 'addr'is supplied. The string is not null terminated.
(E.)	(addr -- ; fs: r --) Format float 'r' as a counted string in engineering format in the buffer whose address 'addr'is supplied. The string is not null terminated.
(F.)	(addr -- ; fs: r --) Format float 'r' as a counted string in scientific format in the buffer whose address 'addr'is supplied. The string is not null terminated.
Precision	(-- u) Return the number of significant digits currently used by (F.), (FE.), (FS.), F., FE., or FS. .
set-precision	(u --) Set the number of significant digits currently used by (F.), (FE.), (FS.), F., ** FE., or FS. .
min-precision	(u --) Set the number of significant digits currently used by (F.), (FE.), (FS.), FE., or FS. .

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.)

Printing

From the source code. Needs development.

PRINTER	(--) Set printing defaults and parameters
PRINT	(--)
FPRINT	(-<name>-) /parse-word $fprint ;
FPRINT	(-<name>-) parse-word $fprint ;
$fprint	(the-name \ message$ fpr$ locHdl -- } type filename a1 to the printer
print-screen	(--) \ print the physical screen print from first visible row to last visible row
print-console	(--) \ print all lines used in screen save buffer
single-page	(--) \ synonym 1page single-page \ synonym one-page single-page
two-page	(--) \ synonym 2page two-page
2print	(--) two-page fprint single-page ;
four-page	(--) \ synonym 4page four-page

4print (--) four-page fprint single-page ;

#print-screen (start_line lines --) \ print a range of lines from saved Forth screen buffer

page-setup (--)

print-multi-page (--)

page-scaled (--)

print-scaled (--)

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 ***

Programming Technique

Using the Integrated Development Environment IDE

Text here

Using Files For Programming

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 (maybe directories are OK?) or file names with spaces such as 'c:\test\my demo\'. Rename that to 'my-demo' or something similar.

Customization and file access:

FLOAD	(-<filename>-) Load (compile) into the dictionary from "filename". For example: FLOAD "c:\data\forth\program.f””
INCLUDE	(-<filename>-) A synonym for **fload.**
NEEDS	(-<filename>-) Load a file if not already loaded. For example**: NEEDS “c:\data\forth\demo-.f".** .
REQUIRE	(-<filename>-) A synonym for **NEEDS.**
REQUIRED	(addr u ---) Load a file from stored string at (addr u) if not already loaded. For example: s” matrix.f” **REQUIRED**.
.PROGRAM	Output the path and name of the running program.
.FORTHDIR	Output the path of the running program.
.FPATH	Show the base paths of all the Forth components loaded and the search path shows the directories from which components have been loaded and the paths from which files maybe accessed. Used in the form: **.fpath “c:\data\forth\application.f”.**
FPATH+	Append a path to the current search paths in the form: **.fpath “c:\data\forth\newapp.f”.**
S"	("<quote>" -- c-addr u) Compile the following text ending in “ (double quote) in line, returning its address and count. .

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

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

Access and using files:

After editing a file in the IDE, hit F12 to compile and execute the contents of the file. The file will be saved and W32F will open. If the file has compiled without error see ‘ok’. Otherwise you’ll see an error report in the form;
0 pick{{
^^^^^^

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.

** add more 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 <word> Adds **<word>** to the dictionary. When **<word>** is executed the dictionary is trimmed to that point.

ANEW <word> If **<word>** exists the dictionary is trimmed to just before **<word>.** Then **<word>** is placed in the dictionary. This sequence is usually used at the beginning of code under development so recompilation avoids redefinitions.

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.

File Search Paths

Win32F maintain a list of directories (folders) (called 'paths') which is searched upon a file reference (FLOAD, INCLUDE, etc). The PATHS support words are located in \scr\paths.f.

For more on search path support see: \doc\paths.htm and paths.f. Some of the support words include:

.program **Display** the program path.

.forthdir **Display** the forth directory.

.dir **Display** the current directory.

.path **(path --)** Display a directory search path list. Note: The path source will be reset for this path.

.fpath **(--)** Display the Forth directory search path list.

chdir **(-<optional_new_directory>- --)** Set the current directory.

FLOAD **(-<name>-)** Load file "name".

INCLUDE **(-<name>-)** Load file "name".

NEEDS **(-<name>-)** Conditionally load file "name" if not loaded.

REQUIRE a synonym of **NEEDS** Forth 200X name for **NEEDS**. May be relative.

\LOADED- **(-<name>-)** If the following file IS NOT LOADED, interpret line.

\LOADED **(-<name>-)** If the following file IS LOADED, interpret line.

LOADED? **(-<name>- -- flag)** True if the following file is loaded. The filename

PATH: Defines a directory search path. The first cell holds a pointer to 2 cells in the user area which are used to handle a search path. The next 260 bytes are reserved for a counted string of a path. followed by null.
At runtime, it returns address of the counted string of a path.

reset-path-source (path --) Points the path-source to the whole path.

fbase-path+ **(-<directory>- --)** Append a directory to the Forth search base path.

fpath+ **(-<directory>- --)** Append a directory to the Forth search path.

program-path-init (--) Initialize the Forth directory search path list. Automatically done at program initialization and when Paths.f is loaded.

Using files for data

For how to create and use files for data see 'File Creation Reading and Writing'.

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 may 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 ".

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 text interpretation or 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.)**

(text) ("ccc<paren>" --) Skip 'text' until the closing ')'. Both '(and)' must be on the same line (before the invisible end-of-line character).

	When parsing from a text file, if the end of the parse area is reached before a right parenthesis is found, refill the input buffer from the next line of the file repeating this process until either a right parenthesis is found or the end of the file is reached.
((text))	Skip **'**text**'** until the closed **'))'**. The closing **'))'** may be on subsequent lines. This is often used to skip over extensive comments.
.(text)	Display **'**text**'** until the following **')'**. Usually used to add narrative or checkpoints within source code files.
****	Skip the remaining text on a line
\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.
Comment:	Skip text until the following **'Comment;'**. i.e. ends in ';'.
0 [if]	Skip the following text until [then]. This usually is used to select code to be optionally executed or compiled. Thus, it can be used to skip comments or code not yet functional. The **'**0**'** may be any Boolean function; zero to skip to an [else] or [then]. If the Boolean is true execution will continue immediately after [if]

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; 'esc' to exit.
WORDS <text>	Display the words in the dictionary containing 'text'
SEE <word>	Disassemble a word from the dictionary (memory).
VIEW <word>	Locate the source code text of the following Forth word (from a file) and display in the Editor's window.
LOCATE <word>	Copy the word's definition from source code file placing it above the command line.
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
CLS	Clear the screen.
RESET-STACKS	Clear both the data and return stacks.
COPYFILE <file>	
FORGET <word>	Remove <word> and all subsequently defined 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.
EXIT	Used in a colon definition, execution will stop and return to the console.
TIME&DATE	Read the computer clock as ms:sec:min:hour:day:month:year i.e 34 55 22 23 12 2020 (22:55.34 Dec. 23, 2020.
.TIME	Display the time as H:M:S
MS@	Get the local time in milliseconds
MS	(n1 ---) pause for n1 milliseconds
SEC	(n1 ---) pause for n1 seconds.
START-TIME	A VALUE holding the computer time at which TIME-RESET was executed. Executing START-TIME MS@ - gives the milliseconds since TIME-RESET. Using TIME-RESET <action> .ELAPSED reports the execution time of <action>.
TIME-RESET	Capture the computer time into **START-TIME**.
TIMER-RESET	Same a **TIME-RESET**
.ELAPSED	Display the time since **TIMER-RESENT** by reading **START-TIME**.
ELAPSE	Used at the start of a command line to time the following code. Used in the form: **ELAPSE <a-word> <a-word> <a-word> enter**
dup-warning-off dup-warning-on	Will suppress a warning message when duplicating the name of an existing word.
sys-warning-off sys-warning-on	Will suppress a warning message upon mis-use (compiling) of a system word such as **[IF], [ELSE] and [THEN].**

sfs

Keyboard Shortcuts

While most editing is done with the mouse and cursor, a very rich selection of keyboard input enhances the editing process. The alpha character may be upper or lower case.

Cont. c (copy), Cont. v (paste), Cont. a (select all) and F12 (save & compile the file being edited) get most of the use.

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.

File Access Details

As each file is loaded a simple dictionary entry is made consisting of a backward pointer to the prior file 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.

Chapter 5 Expanding Your Usage

Text Strings

Text strings may be interpreted or compiled with:

." **text"**	Used in a colon definition. compiles a string. Upon later execution "text" will be displayed. Not usable when interpreting. See '.('.
.(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.
c" text"	(<text> ---) (--- addr) A state smart word. When used in a colon definition, will compile "text" as an in-line, counted string. Upon execution of that word the address of the count byte of "text" will be placed on the stack. If interpreting, the text will be compiled onto a remote buffer and the address of the count byte returned on the stack.
s" text"	**(compile: <text> --- , execute: --- addr count , interpret: <text> --- addr count)** A state smart word. When used in a colon definition, will compile "text" as an in-line, un-counted string. Upon execution of that word the address of the first letter of "text" and its character count will be placed on the stack. If interpreting, text is compiled into a remote buffer and the address of the first letter and count will be placed on the stack.
z" text"	**(compile: <text> --- , execute: --- addr , interpret: <text> --- addr)** A state smart word. When used in a colon definition, will compile "text" as an in-line, un-counted string ending in a null (zero). Upon execution of that word the address of the count of "text" will be placed on the stack. If interpreting, text is compiled into a remote buffer, as an uncounted string, ending in a null (zero), and the address of the first letter will be placed on the stack.
",	(addr let --) Compile string from addr,len as a counted string at **HERE**.
,"	(-<string">-) Compile a counted string delimited by " at **HERE.** Can be used to initialize a counted string array.

Z", (addr len --) Compile the string, addr len as uncounted chars at **HERE** ending in a null (zero).

Z," (-<string">-) Compile a string delimited by " as uncounted chars null-terminated chars at **HERE.** Can be used to initialize an uncounted string array.

((“)) (--- addr) This is a system level word. Use with caution. This word reaches into the calling word at which point a counted string has been compiled. It places the address of the counted string on the stack and increments the return point so that upon return, execution will continue after the counted string. Used in **ABORT”**.

```
: (test")  (("")) ;  \ gets the string address and skips over the string.
: test" compile (test") ," ;  immediate \ compiles the string structure.
: demo  test" AAAAA" ;  yields the address of the counted string AAAAA
```

In this example, into ‘demo’ test” compiles (test”) and compiled the following text as a counted string. When demo is later executed the call to (test”) used ((“)) to load the address of the in-line counted string, increment over that string and return to demo after the string.

String Utility Words

POCKET A short-term text buffer of the input stream.

char WORD Transfer the following text from the input stream to the temporary workspace POCKET, as a counted string until the ending character ‘char’, yielding the address the counted string in POCKET. Ignore leading occurrences of ‘char’. Usually used in the form of BL WORD <text> to parse text from the input stream for execution or compilation. i.e. ‘ text’ will yield the counted string ‘4text’.

NEXTWORD Behaves the same as WORD with the added feature subsequent text will automatically be loaded upon reaching the end of a file line or console line.

COUNT (addr --- addr+1 count) From the address of a counted string, return the address of the (next) first character of the string and the character count. Typically used as **‘addr COUNT TYPE’ or ‘BL WORD COUNT . . .’.**

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

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

```
WCOUNT CODE WCOUNT     ( str -- addr len )  \ word (2 bytes) counted strings
LCOUNT CODE LCOUNT     ( str -- addr len )  \ long (4 bytes) counted strings
```

-TRAILCHARS (c-addr u1 char -- c-addr u2) If u1 is greater than zero, u2 is equal to u1 less the number of chars at the end of the character string specified by c-addr u1. If u1 is zero or the entire string consists of chars, u2 is zero.

-TRAILING (c-addr u1 -- c-addr u2) If u1 is greater than zero, u2 is equal to u1 less the number of spaces at the end of the character string specified by c-addr u1. If u1 is zero or the entire string consists of spaces, u2 is zero.

-NULLS (c-addr u1 -- c-addr u2) If u1 is greater than zero, u2 is equal to u1 less the number of nulls at the end of the character string specified by c-addr u1. If u1 is zero or the entire string consists of nulls, u2 is zero.

SEARCH	(c-addr1 u1 c-addr2 u2 -- c-addr3 u3 flag) Search the string specified by c-addr1 u1 for the (sub)string specified by c-addr2 u2. If flag is true, a match was found at c-addr3 with u3 characters remaining. If flag is false there was no match and c-addr3 is c-addr1 and u3 is u1.
SKIP	(adr len char -- adr' len') skip leading chars "char" in string.
-SKIP	(addr len char -- addr' len') \ Skip occurrences of char BACKWARDS starting at addr, the end of the string, back through len bytes before addr, returning addr' and len' of char.
WSKIP	(adr len word -- adr' len') skip leading words "word" in string.
Lskip	(adr len long -- adr' len') \ skip leading cells "long" in string.
SCAN	(adr len char -- adr' len') \ search for the first occurrence of ‘char’ in string
-SCAN	(addr len char -- addr' len') Scan for ‘char’ BACKWARDS starting at addr, the end of the string, back through len bytes before addr, returning addr' and len' of char.
LSCAN	(adr len long -- adr' len') \ search for the first occurrence of cell ‘long’ in string.
WSCAN	(adr len word -- adr' len') \ search for the first occurrence of word ‘word’ in string
concat	(addr1 addr2 addr3 ---) Concatenate counted strings into addr3. Not part of W32F. See below for the definition..
“CLIP"	(c-addr1 len1 -- c-addr2 len2) Clip string c-addr1,len1 to c-addr2,len2 where c-addr2=c-addr1 and len2 is between 0 and MAXCOUNTED.
PLACE	(c-addr1 len1 c-addr2 --) Place string c-addr1, len1 at c-addr2 as a counted string.
+PLACE	(c-addr1 len1 c-addr2 --) Append string addr1, len1 to the counted string at
"CLIP"	(c-addr1 len1 -- c-addr2 len2) Clip string c-addr1,len1 to c-addr2,len2 where c-addr2=c-addr1 and len2 is between 0 and MAXCOUNTED.
+NULL	(c-addr --) Append a NULL to the counted string.
/STRING	(c-addr1 u1 n -- c-addr2 u2) Adjust the character string at c-addr1 by n characters. The resulting character string, specified by c-addr2 u2, begins at c-addr1 plus n characters and is u1 minus n characters long. If n1 greater than len1, then returned len2 will be zero. For early (pre Nov 2000) versions of W32F, if n1 less than zero, then returned length u2 was zero. /STRING is used to remove or add characters relative to the left end of the character string. Positive values of n will exclude characters from the string while negative values of n will include characters to the left of the string.
COMPARE	(adr1 len1 adr2 len2 -- n) Compare two strings. The return value is: \ 0 if string1 = string2; -1 if string1 < string2; 1 if string1 > string2
STR=	(adr1 len1 adr2 len2 -- flag) \ compares two strings, case sensitive. The return value is: -1 true if string1 = string2; otherwise 0 false. Same as COMPARE 0=
ISTR=	(adr1 len1 adr2 len2 -- flag) Compares two strings, case insensitive. Returns true -1 if strings are equal; else 0 false.

Use of string operations

W32F has a rich selection of words to support searching text strings or numeric values 0..255 in memory. **SKIP, WSKIP, LSKIP, SCAN, WSCAN and LSCAN** are useful to locate a sequence of text characters. If operating on numeric values, remember they are stored low byte to high byte, 'little-endian'.

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. If the character is not found return are the address of the end of the string and zero.

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 'abab' 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 ‘abcdefffffff’ 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.

Concatenation

W32F does not have a string concatenation (joining) word. Here concat that fills that need.

```
: concat ( addr1 addr2 addr3 ) \ join counted strings addr1 addr2 into addr3
  swap count >r over 1+  3 pick c@ + r@ cmove>
  tuck over c@ 1+ cmove   r> swap  c+!  ;
create left ," abcd"   create right ," efghij"  create *concat 30 allot
cr cr  left count type  cr right count type   left right *concat concat
cr  *concat count type
```

Cell Size and Manipulation

In W32F 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 strings have their specific allocation size. Consistent with this, cell size, variables, constants, fetching and storing most often involves 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` Make a dictionary entry for storage.

`5 CELLS ALLOT` Calculates 5x4=>20 and allocates that storage space in memory. Executing **`A-STRUCTURE`** will place the address of the first byte on storage on the data stack.

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. Usually used to address a cell offset from a base address.

`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.

Locating Words And Their Components

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 words 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
BL WORD	(<text> --- addr) Return the counted address of the next word in the input stream.	
FIND	addr --- xt true; or addr false From the counted string at addr return the execution token xt and TRUE (-1) if found. If not found return that counted string address and FALSE (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 address and character count.
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. If the name is not found, show the address.
.ID	(nfa --)	Type a word name from its name fiend address.

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 (in the range 1 . . 511) you may change the default number of threads for a new vocabularies. W32F allows up to any number of vocabularies to be defined and up to 16 to be available for searching at any given time (a limit of the vocabulary stack).

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

Forth words are grouped in vocabularies (called word-lists in the ANS Forth Standard). They are created by VOCABULARY <voc-name>. Vocabularies are searched in the order they appear in a list of active vocabularies (most recent first) called the ‘Search Order’. The search 0rder is cleared by ONLY. A vocabulary is added to the Search Order by: ALSO <voc_name>. If just the vocabulary name is executed (omitting ALSO) that vocabulary replaces the prior named vocabulary in the search order. ORDER lists the current search order. PREVIOUS deletes the most recent (top) of the search order.

The most recently named vocabulary is at the top of the search order (first to be searched) pointed to by the variable CURRENT. The vocabulary followed by DEFINITIONS is the destination for new definitions, pointed to by the CONTEXT variable.

Executing DEFINITIONS established the vocabulary into which new words will be added/defined. The search order has the capacity of sixteen vocabularies.

A typical search order organization might be:

```
ONLY FORTH    ALSO TESTING   ALSO MY-APPLICATION  DEFINITIONS
```

In this example, new words will be added into **MY-APPLICATION**. When words from the input stream are accepted, they will be searched for (in order): **MY-APPLICATION, TESTING** and, finally **FORTH.**

You may compile specific search orders to facilitate vocabulary selections. For example:

```
: CUSTOM-SEARCH   ONLY FORTH ALSO GAME1 DEFINITIONS ALSO DISPLAY2 ;
```

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.

```
   ONLY FORTH  ALSO MY-APPLICATION DEFINITIONS  ALSO TOOLS
```

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

Summary:

ORDER	List the search order.
ONLY	Clear the search order.

<a-vocabulary-name> Executing a vocabulary name places it at the top of the search order.

FORTH Place the vocabulary FORTH as the top of the search order.

ALSO <voc> Add the following vocabulary to the search order.

VOCS List the vocabularies

PREVIOUS Drop the most recent addition to the search order.

DEFINITIONS Establish the most recent vocabulary in the search order for new definition additions.

CURRENT Points to the most recent vocabulary in the search order.

CONTEXT Points to the vocabulary into which new words will be added.

VOCABULARY <name> Create a new vocabulary named <name>. (Default 7 threads.)

n #VOCABULARY <name> Create a new vocabulary <name> with n threads.

GET-ORDER (--- list n) Copy the contents of the search order to the data stack with n being the number of entries. Is used to make a temporary copy of the search order by holding it on the data stack. A clearer and more structured method would be to compile the search order into a definition, as noted above.

SET-ORDER (list n ---) Replace the search order with the list of n entries from the data stack.

Memory Management

W32F uses the resident Windows heap function to supply memory blocks for use in applications. Using it, fixed blocks of memory may be allocated, deallocated and resized. The typical use is to receive the contents of files. W32F includes the ANSI Forth words and versions integrated with the W32F error response using CATCH and THROW. See the W32F Kernel code: Line 3917 onward.

After allocation, the typical application saves the allocation address in a variable or value to allow its access and deallocation by name.

A typical use to create and release 4094 bytes of memory would be (my-mem holds the address of the allocation which also serves as the file identifier:

```
0 value my-mem    4096 malloc to my-mem    my-mem release
```

malloc (u -- addr) Allocate u address units of data space which will begin at addr. This is the W32F version of '**allocate**' in which any error response is handled automatically.

allocate **(** u -- addr ior) Allocate u address units of contiguous data space. If the allocation succeeds, addr is the starting address of the allocated space and ior is zero. Otherwise, ior is true on error. This is the ANS definition at which error control is the user responsibility.

release (addr --) Release the memory space pointed to by addr. This is the W32F version of '**free**' in which any error response is handled automatically.

free (addr – ior) Release the memory space pointed to by addr and return false. Otherwise ior is true if it failed. ANSI version.

realloc (u addr1 – addr2 fl) Replace the memory allocation at addr1 with addr2. Copy the contents of the area starting at addr1 into addr2 up to the limit of the smaller quantity. If the operation succeeds, a-addr2 is the starting address of u address units of newly allocated memory and ior is zero. If the operation fails, a-addr2 equals a-addr1, the region of memory at a-addr1 is unaffected, and ior TRUE. This is the W32F version of the ANSI resize.

resize (addr1 u -- addr2 ior) Change the allocation of the data space starting at the address a-addr1 to u address units. The contents beginning at a-addr1 are copied to a-addr2, up to the minimum size of either of the two regions.

If the operation succeeds, a-addr2 is the starting address of u address units of allocated memory and ior is zero. If a-addr2 is not the same as a-addr1, the region of memory at a-addr1 is returned to the system. ANSI version.

If the operation fails, a-addr2 equals a-addr1, the region of memory at a-addr1 is unaffected, and ior is the implementation-defined I/O result code.

The internal, key components of memory management are: Windows function calls using GetProcessHeap, HeapAlloc, HeapFree, HeapReAlloc, and HeapSize.

File Creation, Reading & Writing

W32F (and the ANS Standard) provide a full selection of words to support file access. Three modes of file access exist: read only, write only and read & write. Two file types are supported: ASCII characters with lines ending in 'cr' and/or 'lf' and binary, with no specific byte/character significance.

The most common file read is to load and compile a Forth program. **fload, include** and **require** will be used for this task. The text file is created in the W32F IDE or other text editor. This is discussed above.

The next most common task is to read and write text files, generally line by line. Line-based file actions use **read-line** and **write-line** processing each line until reaching its ending **cr** (ASCII 13) and/or **lf** (ASCII 10). A typical use would be to process a .csv data file (comma separated value format).

The final, and least used, is to read and write a binary file. That is, each character/byte may have the value from $00 to $ff. For this we use **read-file** and **write-file** which uses the byte offset within the file.

Files are created at zero length and will grow dynamically. The basics for a new file are: 1) Create a buffer area via Windows functions from its 'heap' allocation process (or an array by **create**) and refer to MALO, or else in dictionary space by Forth's **here**. 2) Place data into that buffer, 3) Create a file from its name and access method, noting its file identifier (often called its 'fid or handle'), 4) Copy data from the buffer area into the file, 5) Close the file, 6) Release the Windows buffer area, if used. During this process you may manipulate data in the buffer and /or portions of the file.

Files always grow or shrink relative the last entry located at **file-position**. The words **write-file** and **write-line** add characters. A file may be shorted or lengthened from its end point (**file-position**) by **reposition-file**.

The basics for reading from an existing file are: 1) Open the file from its file name (or numeric identifier 'fileid' or handle), 2) Create a suitable buffer area via Windows functions from its 'heap' allocation process (or space in dictionary memory), 3) Copy some or all of the file into the buffer, 4) Close the file, 5) Release the buffer area, if used. During this process you may manipulate data in the buffer and/or portions of the file.

The 'action point' in the file buffer is the location at which copying, and insertion begin. The first byte in a file buffer or file is number zero. **file-position** gives the current file insertion point. **reposition-file** allows you to change the file's insertion point.

A typical use to create an empty file 'MyFile' could be**:**

```
0 value My-File-ID  (a value to hold the file identifier)
s" A-File" r/w  create-file (returns fileid and ios error code)
-280 ?throw (check ios) to My-File-ID (save the fileid value)
```

Here is an example of reading a data file: The named parameters are created and later the name and buffer size are added:

```
0 value My-File-ID        (holds the file identifier, fid)
0 value My-File-Name      (will point to the name later given)
0 value My-File-Contents  (will point to later created storage space)
0 value My-File-Size      (later determined)
here to My-File-Name ," c:\forth\work\my-file.txt" (a named, counted string )
My-File-Name count r/0 open-file   (returns the file-id and ios)
    -280 ?throw   to My-File-ID    (check error and save the file ID)
My-File-ID  file-size  281 ?throw  to My-File-Size (determine the file size)
My-File-Size here to My-File-Contents allot  (create storage space)
My-File-Contents My-File-Size My-File-ID read-file  (copy to text buffer)
    -281 ?throw     ( and check the error code)
My-File-ID close-file  -280 ?throw  ( file closed)
```

fload (-<filename>-) Load (compile) into the dictionary from "filename".

include a synonym of **fload**

require (-<filename>-) Load (compile) into the dictionary "filename" only if it has not been loaded already.

create-file (addr slen fmode -- fileid ior) Create an empty file named in the character string specified by addr and slen, and open it with file access method fmode (**r/o, w/o or r/w**). If a file with the same name already exists, recreate it as an empty file.

If the file was successfully created and opened, ior is zero, fileid is its identifier, and the action point is positioned to the start of the file. Otherwise ior is true.

open-file (addr slen fmode -- fileid ior) Open the file named in the character string specified by 'addr slen'. If the file is successfully opened, ior is zero, fileid is its identifier, and the action point is set at the beginning of the file. Otherwise, ior is result code and fileid is undefined.

close-file (fileid -- ior) Close the file identified by fileid. 'ior' returns true if the file doesn't exist or false for correct execution.

write-file (addr slen fileid -- ior) Write slen characters from addr to the file identified by fileid starting at its **file-position**. 'ior' is false for correct completion or true if not.

At the conclusion of the operation, **file-position** returns the next action point after the last character written to the file, and **file-size** returns a value greater than or equal to the value returned by **file-postion**.

read-file (addr slen1 fileid – slen2 ior) Read slen1 consecutive characters to addr from the current position of the file identified by fileid.

If slen1 characters are read without an exception, ior is zero and slen2 is equal to slen1.

If the end of the file is reached before slen1 characters are read, ior is zero and slen2 is the number of characters actually read.

If the operation is initiated when the value returned by FILE-POSITION is equal to the value returned by FILE-SIZE for the file identified by fileid, ior is zero and slen2 is zero.

If an exception occurs, ior is the implementation-defined I/O result code, and slen2 is the number of characters transferred to addr without an exception.

An ambiguous condition exists if the operation is initiated when the value returned by FILE-POSITION is greater than the value returned by FILE-SIZE for the file identified by fileid, or if the requested operation attempts to read portions of the file not written.

At the conclusion of the operation, FILE-POSITION returns the next file position after the last character read.

write-line (addr slen fileid -- ior) Write slen characters from addr into the file specified by fileid. 'ior' is false for correct execution and true if not. At the conclusion of the operation, **file-position** returns the next action point after the last character written to the file, and **file-size** returns a value greater than or equal to the value returned by **file-position.**

read-line (addr slen1 fileid – slen2 eof ior) Read the next line from the file specified by fileid into memory at the address addr. The line is expected to terminate with the ASCII cr and/or lf characters but are not included in the slen count. At most slen

characters are read. The line buffer provided by addr should be at least slen+2 characters long.

If nothing is read return slen-0=, eof=false, ior=false.

If the operation succeeded, ior is true, otherwise is zero. If a line terminator was received before slen1 characters were read, then slen2 is the number of characters, not including the line terminator, actually read (0 <= slen2 <= slen1). When slen1 = slen2, the line terminator has yet to be reached.

If the operation is initiated when the value returned by **file-position** is equal to the value returned by **file-size** for the file identified by fileid, eof is false, ior is zero, and u2 is zero. If ior is non-zero, an exception occurred during the operation and ior is the implementation-defined I/O result code.

An ambiguous condition exists if the operation is initiated when the value returned by **file-position** is greater than the value returned by **file-size** for the file identified by fileid, or if the requested operation attempts to read portions of the file not written.

At the conclusion of the operation, **file-position** returns the next file position after the last character read

flush-file (fileid -- ior) Force any buffered information written to fileid to be written to mass storage. If the operation is successful, ior is zero; otherwise true.

delete-file (addr clen -- ior) Delete the file named in the character string specified by addr and clen. ‘ior’ is false or correct execution and true if not..

rename-file (addr1 clen1 addr2 clen2 -- ior) (c-addr1 u1 c-addr2 u2 -- ior) Rename the file named by the character string addr1 clen1 to the name in the character string addr2 clen2. ‘ior’ is false for correct execution and true if not.

included (addr len --) Load (compile) into the dictionary from a file name (path) at addr, len.

"fload synonym of **included.**

include-file (fileid --) Load (compile) into the dictionary from an open file from its "fileid".

$fload (addr --) Load (compile) into the dictionary from the counted string at addr.

“open (addr slen -- fileid ior) Open the file by its name at addr, slen. Return fileid and ior false of executed and false if not.

$open (addr -- fileid ior) Open a file by its counted filename specified by addr. Return its fileid and ior false if the file exists or true if not.

ascii-z (addr clen buff -- buff-z) Make a null-terminated copy of the string at addr clen in buffer buff and return the address of the first character.

file-position (fileid -- len-ud ior) Return the current action point len-ud within the file fileid. 'ior' is true for correct execution. If execution is incorrect ior is true and len-ud is undefined.

reposition-file (len-ud fileid -- ior) Reposition the action point for the file fileid to len-ud. 'ior' is false for correct execution and true if not. At the conclusion of the operation, **file-position** will yield the new value len-ud, the current action point.

**** There is an error in **advance-file, reposition-file, file-append.** These words <u>will</u> produce a stack underflow ***

```
: SetFP        ( parms -- len-ud 0 | len-ud error )
   FPARMS-FP Call SetFilePointer dup -1 ( INVALID_SET_FILE_POINTER ) =
   IF Call GetLastError DUP NO_ERROR =
     IF   DROP SWAP 0                          \ return len-ud 0=success
     else NIP 0 SWAP then                      \ return 0 0 ior=err
   else
     SWAP 0                                    \ return len-ud 0=success
   then ;

: advance-file ( len-ud fileid -- ior ) \ RELATIVE position file, not ANS
 \ ior - 0 = success
   FILE_CURRENT SetFP ( nip ) nip ;

: reposition-file ( len-ud fileid -- ior ) \ ior - 0 = success
   FILE_BEGIN SetFP ( nip ) nip ;

: file-append   ( fileid -- ior ) \ ior - 0 = success
   0 0 rot FILE_END SetFP ( nip ) nip ;
```

file-append (fileid -- ior) \ ior - 0 = success 0 0 rot FILE_END SetFP nip nip ; ???

file-size (fileid -- len-ud zero ior) 'len-ud' is the size, in characters, of the file fileid. 'ior' is false for correct operation and true if not. This operation does not affect the value returned by **file-position.** 'len-ud' is undefined if ior is non-zero.

*** W32F **file-size** in incorrect. A corrected version follows. GetFileSize returns the file size, if correct or false (-1) upon an error. In addition a check is made using GetLastError ***

```
: file-size     ( fileid -- len-ud ior )
    sp@ swap                        \ s-address and file-id
    call GetFileSize                \ size | -1 upon an error
    dup INVALID_FILE_SIZE =         \ size size_eror_flag
    call GetLastError NO_ERROR <>   \ size size_error_flag last_error_flag
      or  ;                         \ size true_on_error
```

resize-file (ud fileid -- ior) Set the size of the file fileid to ud. ‘ior’ is false if the file exists and true if not.

If the resultant file is larger than the file before the operation, the portion of the file added as a result of the operation might not have been written.

At the conclusion of the operation, **file-size** will return the value ud and **file-position** will return an unspecified value.

file-status (addr slen -- x ior) Return the status of the file identified by the character string at addr slen. If the file exists, ior is zero otherwise true. ‘x’ contains implementation-defined information about the file.

fsave-file (addr slen filename --) Create a file named filename (from the address if its counted string) with the contents at addr, slen; conclude by writing and closing the file.

refill (-- flag) When the input source is a text file, attempt to read the next line from the text-input file. If successful, make the result the current input buffer and return true. Otherwise return false.

s" Executing: ("ccc<double-quote>" -- addr slen) Place the following string (uncounted), ended by “ (double-quote), into a temporary buffer and return its address addr and character count slen. The capacity of the temporary buffer is 260 characters. Subsequent uses of **s"** may overwrite the temporary buffer.

Compilation: ("ccc<quote>" --) Compile the string ‘ccc’ delimited by " (double-quote) that, at run-time, will yield its address addr and character count slen.

bin In W32F bin has no action; it is provided for compatibility with other programs specifying binary access. The file content-based access method is selected by the choice between line-based and file-based read and write.

r/o (-- fam) fam is the value specifying the read-only file access method. **$80000000.**

w/o **(** -- fam) fam is the file access method value for selecting the write-only file access method. **$40000000.**

r/w **(** -- fam) fam is the file access method value for selecting the read/write file access method. **$C0000000.**

Notes on file access and corrections are needed.

Correct operation: **open-file, close-file, create-file**.

The current W32F **file-size** returns (size false false) which is incorrect. It should return (size lior). See corrected code, above.

The following Windows functions are called by W32F file words. **CreateFile, CloseHandle, ReadFile, WriteFile, DeleteFile, MoveFile, SetFilePointer, FlushFileBuffers, SetEndOfFile, GetFileSize.**

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

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

n SYS-ALLOT n APP-ALLOT allocate n bytes in the memory space. (**ALLOT**)

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

b SYS-C, b APP-C, compile b (8 bits) into the next available memory byte. (**C,**)

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

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

Advanced Topics

Structure of Dictionary Fields (as of 2003)

Word name structure in high memory, above C0000h

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

Word code and parameter field in low memory.

[cfa field]	+0	CFA code field address, pointer into user's dictionary space.
[body field]	+4	PFA parameter field address, pointer into user's dictionary space.

Original Dictionary Fields (1995)

Name	Field Bytes	
alignment bytes	0..3	filler bytes to align optimize field to the cell boundary.
optimize field	4	
alignment bytes	0..3	filler bytes to align view field
name	1..n	ascii text of a word name
name length	1	number of bytes in the word name (nfa)
view field	4	view field, line number offset into source code file (vfa)
link field	4	points to the prior word's link field (lfa)
c.f. pointer	4	points to code field in user's dictionary space (cfa)

Structure of Data Fields

Name Field	Byte size	
code field	4	points to its machine code interpreter.
param field	4+	or more bytes. Used for data storage, machine code or a sequence of execution token addresses (colon definition).

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 (link field) to match the input text against each word's name. The specific linked list (using a chain of link field pointers) is selected by a hashing algorithm. The search routine searches toward low memory, through the earlier defined words stopping at each link. At each link, the name length in the name length 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 and a 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 word 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.

Name	Action	Function
`	(--- cfa)	single apostrophe, pronounced 'tick'
>BODY	(cfa -- pfa)	from the code field address get the parameter field address.
>NAME	(cfa -- nfa)	from the code field address get the name field address.
>VIEW	(cfa -- vfa)	from the code field address get the view field address.
>OFA	(cfa -- ofa)	get the optimization field address
BODY>	(pfa − cfa))	from parameter field address get the code field address.
NAME>	(nfa -- cfa)	from the name field address get the code field address.
.NAME	(cfa ---)	Display the word's name otherwise show address.
?.NAME	(cfa ---)	Display the words' name.
.ID	(nfa ---)	Display the word's name
N>LINK	(nfa -- lfa)	from the name field address ger the link field address.
N>OFA	(nfa -- ofa)	from the name field address get the optimize field address.
N>CFAPTR	(nfa -- cfa-pointer)	
VIEW>	(vfa -- cfa)	from the view field address get the code field address.
L>NAME	(lfa -- nfa)	from the link field address get the name field address.
LINK>	(lfa -- cfa)	from the link field address get the code field address.

VLINK>VOC	(voc-link-field -- voc-address)
VOC>VLINK	(voc-address -- voc-link-field)
VOC#THREAD	(voc-address -- #threads)
VCFA>VOC	(vocabulary-cfa -- voc-address)
VOC>VCFA	(voc-address -- vocabulary-cfa)

Advanced information on word internal structure

In this version of Win32Forth the following **specific 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 user variable
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

CREATE , DOES> AND ;CODE

Not only does Forth contain words to create words **(: (colon), 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 assembler op-code definer would appear as:

: OP-CODE CREATE , DOES> @ + W, ; with its usage below.

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

Level 2: When a 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 ' , ' (two commas) and exits. When the new 'DOUBLE-FALSE' runs its execution begins after ';CODE'. It extracts the values compiled by ' , ' (two commas) and places them on the stack. It then returns to the inner interpreter by 'next'.

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 or to reach NEXT to continue interpretation. Otherwise, the Forth system crashes.

Reusing Word Names

Forth word names may be reused. 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 newer definition will be compiled and used from the point of its creation forward.

During the definition (compilation) of a new word in the dictionary, its current, partial definition is blocked from Forth's FIND word. Thereby, a word name may be reused and the prior definition of, say, LINKER can be used in a new definition of LINKER.

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.** **REVEAL** will transfer it to the **CONTEXT** vocabulary for ongoing usage.

Manipulation of memory

Note: all of the following operate over bytes. To operate over memory 4 byte cells properly place CELLS to adjust 'n' to the corresponding number of 4 byte cells.

ALLOT	(n ---) allocates 'n' bytes 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 'to' address. This word selects between CMOVE and CMOVE> to avoid over-writing unmoved bytes.
FILL	(addr n char---) Fill memory from addr to addr+n-1 'char'.
ERASE	(addr n --) Clear the addressed bytes to zero/
BLANK	(addr n --) Fill the addressed bytes with the ASCII blank character $20

Input 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.

Conditional Execution and Commenting

Several forms are provided to skip over comments and documentation:

[IF] [ELSE] [THEN]	conditional execution, see below
\s	skip text until the end of file.
****	skip text until the end of the line
//	skip to end of line
--	skip to end of line
(comment)	skip text on the same line until the closing **)**
((multiple lines))	skip until **))** or reaching the end of file
.(to be printed)	print the comment area during compilation
/* comments */	comments over multiple lines.
(* comments *)	comments over multiple lines.
DOC <documentation> ENDDOC	skip intermediate text
\- <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. Forth 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

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.

Floating Point Summary

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, non-destructively.
`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	Plasce 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.

(The following, clarify ???)

F!, F@, FVARIABLE, FDUP The usual floats on the floating point stack

SF!. SF@. VARIABLE, DUP 4 bytes on the data stack.

DF!, DF@, 2VARIABLE, 2DUP8 bytes on the data stack.

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 compiled 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.

Chapter 6 Advanced Topics

Advanced Topics, assembler, multitasking, user area, object oriented programming.

Debug, Tracing Word Execution

Win32Forth includes a debugging facility similar to the debugger in F-PC. You can use the '?' (question mark) key while debugging to see a list of the available debugging commands.

To select a word for debugging, use the word DEBUG as follows (let us debug the Forth word WORDS) :

DEBUG WORDS [enter] The debugger installs a breakpoint into the first cell of the definition WORDS , and waits for WORDS to be executed. It also sets the value of BASE which the debugger will use to the current base. Since we haven't executed WORDS yet, nothing more happens.

NOTE: As soon as you perform the following suggested command, the debugger will automatically tell the editor to display the source for WORDS . To return to WINED, use Ctrl-PgUp.

To get the debugger to start, you need to execute the word where the breakpoint is set like this;

WORDS ! [enter] The debugger will be invoked, and it will display the stack on entry to WORDS , and the source for WORDS will be displayed in the editor. Return here as described in the note above.

Following the stack display (probably the word '**empty**'), you will see **'const 0**' which means CONSTANT with a value of zero. If you press [enter] while the console window is selected, you will see a [1] 0 displayed indicating one stack item with a value of zero. The debugger then displays TO WORDS-CNT which is where the zero is about to be stored, into the value WORDS-CNT . You can press the '?' key at this point to see the commands available in the debugger. Specifically the commands 'N' (nest) and 'U' (unnest) are useful for stepping into or out of definitions while debugging to get to the point where you can see what is really happening in your program.

Another debugging word is DBG ; it works like DEBUG , but it immediately invokes the word following. In this case you need to setup whatever stack arguments before using it. Of course, you also need to do this with DEBUG as well, before the word being debugged gets executed. Here is an example;

DBG WORDS ! [enter] The debugger starts immediately debugging WORDS , the character '!' is passed to WORDS in the input stream as a substring parameter.

The debugger commands are shown here. They are case insensitive..

Command Action

DEBUG <a_word> [enter] Begin debugging upon execution of <a_word>.
DBG <a_word> [enter] Immediately begin debugging <a_word>.
[enter] Single step execute your program.
[space] Single step execute your program.
[esc] Quit, unbug, abort to Forth
^-PgUp Return to WINED
. (dot) Display the depth of the floating point stack

C	continuous step program execution until a key press
D	Done debugging, run the program to completion
F	Allow entry of a Forth command line
H	Toggle the display between Hex and Decimal numbers
J	Jump over next Word, useful for exiting LOOPs
N	Nest into a colon or does> definition
O	Display the current vocabulary search order
P	Proceed to initial breakpoint location again
^P	Proceed to the current program point again
Q	Quit, unbug, abort to Forth
R	Show the Return stack functions
U	Unnest to definition above this one
V	Display the current vocabulary search order
W <text>	Accept the following text into the 'watch' buffer. That Forth source will be evaluated after each word is traced. Could be used to fetch contents of a variable, store a value, etc.

NOTE: The words **WITHOUT-SOURCE** and **WITH-SOURCE** can be used to turn source code display off and on, respectively. If you have limited memory, you may need to use **WITHOUT-SOURCE** to turn off source level debugging.

See the Help system for information on debugging methods.

Splitting The Top of Stack Value

The data stack values are 32-bit, 4 byte '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.

Display Control GOTOXY

Win32F provides three words for cursor control of the display. With these you can clear the display, place the cursor and read the cursor position. These could be helpful in creating a custom form display using text. This is not full graphic display.

cls	(---) clear the screen. Executing **cls** alone, you will see 'ok' at the top left of the screen.
gotoxy	(column row ---) Place the cursor at column, row. Values are zero based, so 0 0 is the top left corner of the screen. ***** Note**: apparently the first time **gotoxy** is called after program loading an incorrect value is used. To correct for this, upon the first use, enter **0 0 gotoxy** to properly initialize.

getxy (--- column row) Place the cursor's row and column position on the stack, again, zero based.

Lists

W32F provides utilities to create and use a linked list of messages or operators: 1) **throw-msgs** holding the responses to **THROW/CATCH** (See Standard Messages, above), 2) a list of active disk files, 3) A list of deferred words **defer-list**, 4) A list of processes (Windows calls) [**winproc-link**] and 5) A list of library functions [**winlib-link**].

A caution: If your application adds error messages, the kernel **throw-msgs** is modified within the loaded kernel image. If your application is restarted (generally during testing), the original value of **throw-msgs** must be restored.

A typical sequence is:

```
throw_msgs @ constant fix-msgs  \ note the original throw_msgs value
anew my-application             \ define a restart point
fix-msgs throw_msgs !           \ restore the message list after restart
```

The Standard Message list is a singly linked list headed by the variable **throw-msgs** and terminated by a zero link value. The list is structured:

variable **throw-msgs**	Points to the forward link of the first message.
forward-link	Points to the link of the following message (zero in last entry).
message-number	An integer in one cell
message	Text of the message as a counted string.

To add a message 999 in **throw_msgs**:

```
throw_msgs   link,   999 ,  ," Text for 999."
```

message (n ---) For a non-zero n, from the messages of **throw_msgs:** If interpreting from the console, display the error number and message. If from a file, display the most recent line of text underlining the word being interpreted, the error number, the message, path and file name. Then continue execution without a restart.

_message (n ---) a synonym for message.

link, (list --) Add a link in the dictionary (i.e. at here) to the head of the list.

," (-<string">-) Compile a counted string delimited by " at **here**. Can be used to initialize a string array. The string will be padded with nulls to end on a cell boundary.

do-link (input(s) word-xt list-head --- output(s)) Apply the function xt in turn to each element of the list originating at list-head. 'Input' and 'output' are the stack input(s) to and output(s) for xt. Normally the number of inputs equals the number of outputs as they will be used along the list.

Usage: input_parameter(s) ' xx link-head **do-link**

The above follows links; for each link executes the word xx. xx must have a stack picture
([parms ...] link -- [parms ...]). Safe to use even if xx destroys next link and can be used recursively.

add-link (addr list --) Add a link to the head of a list.

append-link (addr list --) Append a link to the end of a list.

un-link (addr link -- f1) Unlink addr from list. f1 is 0 if addr was removed from list or non-zero if addr was not in the list.

DEFERed words

Usually, a Forth word must exist before it can be compiled into another definition. DEFERring words allows: 1) A word may be compiled into a definition before it is actually defined and 2) the execution code for a word may later be changed. First, the word is defined by the word **DEFER**. Later, the actual definition is created. Finally, the execution code of the later word is assigned to the first definition. The final definition must have a different name from the first. This 'two word' method eliminates ambiguity in the dictionary search process. The usual practice is the later definition begins with '_' (underscore). Example:

```
defer textA                 \ the placeholder definition
: _textA  cr cr ." Here is text A ;  \ The final code
: _textB  cr cr ." Here is text B ;  \ the default value
' _textA is textA           \ ' (tick) finds _textA and 'is' assigns it.
' _textB is-default textA   \ textB is assigned as the default
.deferred                   \ view the list of deferred words
```

The first line creates a placeholder word **textA**. At this time it would execute **noop** (no operation). The second line is the actual code for the deferred word. The third line is the code for the default value. The fourth line locates the execution token (xt code field address) of **_textA**. **'is'** assigns that execution token to the originally deferred **textA**. The next assigns the default value.

If the assignment for a deferred words is done in a colon definition, the syntax is:

```
defer textA                          \ the placeholder definition
: _textA  cr cr ." Here is text A ;   \ The final code
: assign-textA  ['] _textA is texta ;  \ the assigning word
assign-textA                          \ make the assignment
.deferred                             \ view the list of deferred words
```

A deferred word may have new code assigned repeatedly. Thereby, specialized error messages maybe assigned for different applications or application portions.

The deferred words appear in a linked list, each with their execution token (xt) and a default execution token (xt). The first entry will be executed for that word. The default value may be restored as an alternative. Upon creation, both values for a deferred word are assigned **noop** (Forth 'no operation').

' (tick) (--- xt) Return the code field address (execution token xt) for the following word.

['] (bracket tick) (compiling: ---) (executing: --- xt) (Used in a colon definition, find and compile the execution token of the following word as a literal value. At run time that xt will be placed on the stack to be passed to **'is'**.

is (xt -<name>-) Set <name> to execute the code of execution token xt.

.deferred (---) Display all of the deferred words in the **defer-list** with their current and default word names.

is-default (xt -<name>-) Set xt into the default field of a deferred word <name>. Compiling **is-default** appears to have a problem and should not be used until this is resolved.

restore-default (-<name>-) Reset the execution for <name> to its default xt.

action-of ("name" -- xt) Return the xt the deferred word “name”. When compiling put into current definition.

The structure of a deferred word is:

defer-list addr	addr is link to the first word in the deferred word list.
word-header	
xt, execution token	4 bytes
link to next deferred word	4 bytes
xt, default execution token	4 bytes

A portion of the list of deferred words, using **.deferred** appears as:

Deferred: **GOTOXY**	does: **C_GOTOXY**	defaults to: **2DROP**
Deferred: **?CR**	does: **C_?CR**	defaults to: **DROP**
Deferred: **CR**	does: **C_CR**	defaults to: **D_CR**
Deferred: **TYPE**	does: **C_TYPE**	defaults to: **D_TYPE**
Deferred: **EMIT**	does: **C_EMIT**	defaults to: **D_EMIT**
Deferred: **CLS**	does: **C_CLS**	defaults to: **NOOP**

Error Recovery, Abort, Abort”, Catch & Throw

W32F provides four methods to detect, report and recover from errors: 1) an unconditional soft restart (**abort**), 2) A conditional report and soft restart (**abort”**), 3) A direct recovery to a known point (**catch/throw**) and 4) A conditional recovery to a known point (**?throw**).

Abort and **abort”**

abort (--) Unconditionally perform a soft restart, clear the stacks and begin interpretation from the console.

abort" (flag -<ccc>- --) If flag is true (non-zero) display the following message <ccc> and **abort**. Can be used as:

```
true  abort” A response to true”
```

nabort! (addr flag --) If flag is true (non-zero) display the message from the counted string at addr and **abort.** Can be used in the form:

```
create astring ," My error message"  astring 1  nabort!
```

abort! (addr --) Unconditionally display the message at the counted string addr and **abort**. Can be used in the form:

```
create astring ," My error message"  astring   abort!
```

Catch and Throw

Words **catch** and **throw** provide a method to recover from an error (unexpected condition) in a controlled way. **catch** denotes a recovery point. **throw** will selectively return to that point based on a stack value. For example, **catch** generally activates a top level, text interpreter or control loop (in the case of automation) followed by selections to be made from an error code sent by **throw.** A zero value thrown gives no action. The corresponding later THROW must appear in words called by the word containing CATCH.

CATCH and THROW may be nested and are reentrant. That is, for **CATCH . . . CATCH . . . THROW . . . THROW** the outer **CATCH/THROW** and in inner **CATCH/THROW** are paired.

In the W32F, **CATCH** is located in the outer, text interpreter and will be paired with any **THROW** or **?THROW** later used.

In applications, **catch** would be located in a special purpose text interpreter (modeled after **quit**). In a control system, **CATCH** would be located in the outer control loop and subsequent **THROW** or **?THROW** used within application words called below that outer control loop.

System words use negative error codes. Application programs are to use error codes greater than zero (positive integers).

In the absence of any user's **catch**, if a positive, non-zero value is thrown in a Win32F application, execution will show the error number and return to the outer text interpreter **quit** (which contains the system's **catch**). If one of the defined negative values is thrown the Win32F error message will be shown. i.e.

-22 throw you will see: **Error(-22): THROW control structure mismatch**

Catch sets up a selective recovery point and response. It is used in the form:

```
: outer   . .  .  [‘] outer-loop catch <error response> <restart)  ;
: b-word  . .  .  if -280 throw then  .  .  .  ;
```

The **[‘] outer-loop** is generally a loop structure, outer control loop or text interpreter. In the word **outer**, **outer-loop** calls all lower level code. Control only passes from **outer-loop** to <error response> upon a lower lever t**hrow** or **?throw**. The outer loop in W32F is **quit**.

If a true (non-zero value) is passed to a subsequent (lower level) **throw**, control returns just after '**catch**' for any report and response. The non-zero stack value (in this case -280) is passed to that response and execution continues from that point.

A convenient form is ?throw. It accepts a Boolean value and an error code in the form

```
flag  -280  ?throw .  .  .
```

If **flag** is true, then **?throw** passes the value -280 to **catch.** For a zero (false) flag no action is taken.

CATCH (cfa -- n) Execute the word given by its cfa which should be a closed loop structure. Upon a subsequent **throw** execution exits that loop structure and continues after **catch** with the received parameter 'n'. Be aware: any parameters for the 'cfa word' are still on the stack, under 'n'.

THROW (n ---) Begin a possible error response. If 'n' is zero no action is taken. If 'n' is non-zero terminate execution at this point and return to just after the most immediate CATCH for error reporting and a restart.

?THROW (flag n ---) If flag is no-zero pass the value 'n' to **throw**. Otherwise, no action.

Standard Messages

The Forth ANS Standard specifies some pre-defined messages for **throw** and **catch**. This list is abbreviated. See the ANSI Standard for the full list in Section 9.3.5.

The system numbers assigned to CATCH and THROW are:

ANS Standard Values:

Error number	Message text
-1	no message
-2	message from ABORT". High-light the offending source code test.
-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, execution continues.

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

Execution Chains

Chains are lists of Forth words, each list having a related function, such as numeric input conversion. These words are linked by a sequence of forward address pointers ultimately ending in a value of zero. Calling an execution chain will execute all the words in that particular list (chai).

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?

Chain word summary

New-chain

Chain-add

.chain (chain ---) Display the word names in a particular chain.

.chains Display all the chains and their word contents. : The two sections are System Chains and Application Chains.

\ Execution chain words

(.chains) (link --) \ Display the contents of a specific chain.

The W32F chains are:

initialization-chain \ chain of things to initialize
unload-chain \ chain of things to de-initialize
forth-io-chain \ chain of things to do to restore forth-io
ledit-chain \ line editor function key chain
reset-stack-chain \ chain for stack reset
msg-chain \ chain of forth console Windows' messages
in-system
semicolon-chain \ chain of things to do at end of definition
forget-chain \ This is a chain of types of things to forget. It is not a command to 'forget'.
post-forget-chain \ chain of types of things to forget
pre-save-image-chain \ chain for things to be done to an image prior to saving
\ new chain for reset-stack-chain added \ January 22nd, 2004 - 13:53 dbu
reset-stack-chain chain-add _RESET-STACKS

Compilation Checking ('Security')

Forth performs a number of error (security) checks, most often when compiling or for file access. At control structures such as IF and BEGIN check values are placed on the stack. At the corresponding THEN or UNTIL an error will be reported if the check values do not match.

A check of stack height is made at the start and end of compiling a colon definition (':'). If any values are added or deleted an error advisory is given. To suppress this report place NOSTACK1 at the end of the line.

A warning will be given when a word name is duplicated in a subsequent definition in the sane vocabulary. 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

*** [more sys-error words here] ***

Recursion

Some mathematical processes can use their own execution function repetitively, called Recursion. As the name of a word is not available while being defined (see Renaming Words) Forth provides the word RECURSE for this function. Use of RECURSE will compile at its location a call to the Forth word being defined (itself). Thus:

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

will place the numbers 0 through 10 on the stack.

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'**

Multi-tasking

[This needs work to sort out the user area (permissive multi-tasking support) versus pre-emptive multi-tasking. See Win32Forth/src/lib/task.f\

W32F provides the core elements for multi-tasking. Each task has its own set of User Variables dedicated to that task. The typical use of the full user area for the console task (Forth's user interface) and then abbreviated user areas for each additional task that do not have console interaction. Traditional Forth's have a permissive, round-robin multi-tasker using a work **pause**. Applications must periodically release control to the multi-tasker. In W32F the preemptive multi-task capability of Windows is used. No '**pauses**' are required. True?

Each task requires the declaration of memory for a user area. Added tasks are linked through their user areas. The TASK-STATUS holds those control parameters. One is the run/idle status. Another is the Forth address of the next task's user area base value. When the multitasker is running, in rotation, each task STATUS flag is checked. If TRUE (?) execution of that task is resumed. If FALSE execution jumps to the next tasks user area. This is a round-robin process. Each running task must contain at least one WINPAUSE to pause its execution and pass control around the other linked user areas. ?Is this true??

The Task Control Block

The task control block (also known as task-block or TCB) is a small structure either allotted in the dictionary or allocated on the heap containing information about a task. The xt and parameter variables are set when the task-block is created. The stop flag can be set by other tasks and is used to signal the task that it has been asked to finish.

The ID is set when the task is created and is valid only until the task terminates. The handle is set when the task is created and is valid until it is closed by the API CloseHandle function, even after the task has terminated. The operating system does not free the OS resources allocated to a task until all handles (except for the pseudohandle returned by the API GetCurrentThread) are closed

and the task has terminated. Programs should close the handle as soon as it's no longer needed (if it's never used close it at the start of the task word).

The User Area

When a task is created the operating system allocates a stack for the task. Win32Forth splits this stack into three regions, a return stack, a User area and a data stack. The address of this User area is stored in thread local storage so that callbacks have access to the correct User area for the task. (Versions prior to V6.05 always used the main task's User area for callbacks).

When a task starts the contents of the User area are undefined except Base is set to decimal. The exception handler is set so the task exits if an exception is thrown, returning the error code to the operating system. TCB is set to the task control block of the task. RP0 is set to the base of the return stack. SP0 is set to the base of the data stack. All other User variables used by a task should be explicitly set before use. If the task uses floating-point words then FINIT should be called first.

Multi-tasking Glossary

All are contained in lib/src/task.f

exit-task	(n --) Exit the current task returning the value n to the operating system, which can be retrieved by calling GetExitCodeThread. The stacks and user area for the thread are freed and DLLs are detatched. If the thread is the last active thread of the process then the process is terminated.
create-task	(task-block -- flag) Create a new task which is suspended. Flag is true if successful.
run-task	(task-block -- flag) Create a new task and run it. Flag is true if successful.
suspend-task	(task-block -- flag) Suspend a task. Flag is true if successful.
resume-task	(task-block -- flag) Resume a task. Flag is true if successful.
stop-task	(task-block --) Set the stop flag of the task block to true.
task-block	(parm cfa-task -- addr) Build a task block in the dictionary, initialize the parameter and xt and return the address of the block.
task-stop?	(task-block -- flag) Flag is true if stop-task has been set by another task. In this case the task should do any necessary clean-up and exit.

Locking Resources

Since the multi-tasker is pre-emptive it is sometimes necessary to restrict access to resources to a single task to prevent interference between different tasks. Win32Forth provides a set of words for efficiently locking sections of code. The system also contains some locks used internally that are transparent to the user.

lock	(lock --) If another thread owns the lock wait until it's free, then if the lock is free claim it for this thread, then increment the lock count.
Unlock	(lock --) Decrement the lock count and free the lock if the resultant count is zero.
Trylock	(lock -- fl)
init-lock	(lock --) Initialize a lock

make-lock (compiling: "name" -- runtime: -- lock) Create a new lock. When executed the lock returns it's identifier.

init-locks (--) Initialize all the locks.

init-system-locks (--) initialize system locks for multitasking

Forgetting Locks

Before using FORGET or executing MARKER words, unlock any locks which are about to be forgotten to avoid memory leaks AND exit any threads which will be forgotten to avoid CRASHING !! YOU HAVE BEEN WARNED

See Win32Forth/src/lib/task.f

Multi-tasking Support Words

TASK-STATUS (32 bits for miscellaneous purposes. The lower 24 are reserved for the system)

RP0 Initial return stack pointer.

SP0 initial data stack pointer)

SP@ place the memory address of the stack of the element prior to the location of the SP@ value. That, is

TCB Ttask control block

(*1) absolute min user area)

&EXREC exception handling)

&EXCEPT for exception handling

exc-guard

exc-access

HANDLER throw frame

LP local variable pointer)

OP object pointer

BASE numeric radix

RLLEN read line length, used in file i/o see read-line

RLNEOF read line not EOF, used in file i/o see read-line)

HLD numeric output pointer)

80 CHARS ALIGNED + (numeric output formatting buffer)

PAD user string buffer)

NEXT-USER ! Save top user variable offset in NEXT-USER.

CONSTANT USEREXTRA Add to USERMIN if you are going to do I/O.

CONSTANT USERMIN Absolute min user area.

See KERNEL.F at line 1990 for the User Area layout and the basics of multi-tasking.

User Work Space

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 8 bytes each.

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. SDFS
DEFINED <word>	(-- str 0 | cfa flag) Search for the following <word>. If found give its code field address and a true flag. If not found give the counted string address of <word> and a false flag. The search will be performed in upper case.
WITH-ADDRESS	If 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:

.S Display and maintain the contents of the data stack

.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.

RANDOM (n1 --- n2) Generate n2 which is a random number between zero and n1. n2 will have the same sign as n1. *** Caution *** The generator randomness deteriorates above n1 = 40,000 and must not be used above that value. That is, the output distribution deviates by more than about 3% per decile (each tenth of output range) in randomness. That this generator is randomized upon W32F start up suggests the generator results are not deterministic (should not repeat run to

run.). However, this has not been verified and there are many pitfalls in random number generation.

RANDOM-INIT (---) Initialize the random number generator from the real-time clock. This is automatically done when W32F is started.

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.

SAVE-INPUT and RESTORE-INPUT

The input stream from the console or mass storage is located by several system variables. If you wish to read from that stream, advance and then restore to a noted original location use SAVE-INPUT and RESTORE-INPUT. These typically would be used to read the next word in the input stream, process it and then back up to process that text again.

SAVE-INPUT **(--- n1 … nx)** Save the parameters specifying the current input stream.

RESTORE-INPUT **(n1 .. nx --- flag)** Attempt to restore the input source specification to the state described by x1 through nx. flag is true if the input source specification cannot be so restored.

```
: TEST     SAVE-INPUT   CREATE  RESTORE-INPUT
           ABORT" couldn't restore input"  ;
TEST  ABCD . ( to see the address of the newly defined word)
```

Upon executing **TEST ABCD** the word **ABCD** will be created and the input stream restored followed by the test by **ABORT"** to verify the restore was successful. At that point the input stream has been backed up to before the text **ABCD**. The text **ABCD** will be encountered again. As it has just been

defined by **CREATE** it will execute leaving its parameter field address on the stack to be displayed by '**.**'. In this case **TEST** creates a word (**ABCD**) and immediately executes it. Clever but not useful.

```
: TEST save-input 222 constant
       restore-input  abort" bad restore"  ' execute  dup + . ;
TEST demo    and see 444
```

For the above **TEST**: Upon executing **TEST demo**, the current input stream parameters are saved, then **222 constant** creates the constant '**demo**' assigning its value as 222, the input stream is returned to the noted point, the text interpreter encounters '**demo**' which is found in the dictionary using "'" (tick), it is executed placing 222 on the stack, it is doubled and displayed as the value 444. Again, **TEST** creates a word and then immediately uses it. This is a valid but not particularly useful construct.

```
: action   target definitions  save-input <input stream code>
           host   definitions  restore-input abort" bad restore"
           <repeating the input stream code> ;
action     ?csp !csp compile (;code) init-asm ;
```

For the above '**action**': add definitions to the 'target' vocabulary, mark the input stream, execute/compile code from the input stream until ';', add definitions to the 'host' vocabulary, restore the input stream and again execute/compile from the input stream until '**;**'. This construct is often used in target or cross compilers to place the same code sequence in two vocabularies. Depending on the other words present the re-compiled sequence may have differing actions. For example, one might write to disk while the other writes to ram memory.

DOS Commands ** there are more DOS commands with . ??

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.

editor" (-<string">-) Set the editor command string.

browse" (-<string">-) set the browser command string

shell" (-<string">-) set the shell command string

dos" (-<string">-) set the dos command string.

.shell Display the editor, browser, shell & dos strings:

An example of using .shell:

```
.shell      EDITOR" Win32ForthIde.exe /e %LINE '%FILENAME"'
            BROWSE" Win32ForthIde.exe /b %LINE '%FILENAME"'
            SHELL" CMD.EXE /c "
dos" cmd.exe "
```

Object Oriented Programming

A suite of object oriented programming exists on top of Win32F. This provided the functionality for the graphic display, menus, Editor and Integrated Development Environment IDE.

Neon-style Object Oriented Programming

This guide on Win32Forth Object Oriented Programming, OOP was from VFX Forth documentation which was derived from Andrew McKewan's original.

The path to this was from an ANS OOPs package which has its origins in Neon, an object oriented language derived from Forth for the Macintosh. It was converted to an ANS package by Andrew McKewan, and ported to VFX Forth and optimized and extended by MPE. The text below is based on that provided by Andrew McKewan

Why do this?

When I first began programming in Forth for Windows NT, I became aware of the huge amount of complexity in the environment. In looking for a way to tame this complexity, I studied the object-oriented Forth design in Yerk. Yerk is the Macintosh Forth system that was formerly marketed as a commercial product under the name Neon. It implemented an environment that allowed you to write object-oriented programs for the Macintosh.

While much of Yerk was Macintosh-specific, the underlying class/object/message ideas were quite general. What I hope to accomplish here is to provide any ANS Forth System the ability to use the object-oriented syntax and programming style in these platform-specific systems. In doing so, I have sacrificed some performance and a few of the features.

Object-oriented concepts

The object-oriented model closely follows Smalltalk. I will first describe the names used in this model: Objects, Classes, Messages, Methods, Selectors, Instance Variables and Inheritance.

Objects are the entities that are used to build programs. Objects contain private data that is not accessible from outside the object. The only way to communicate with an object is by sending it a message.

A Message consists of a selector and arguments. When an object receives the message, it executes a corresponding method. The arguments and results of this method are passed on the Forth stack.

A Class is a template for creating objects. Classes describe the instance variables and methods for the object. Once a class is defined, you can make many objects from that same class. Each object has its own copy of the instance variables, but share the method code.

Instance variables are the private data belonging to an object. Instance variables can be accessed in the methods of the object, but are not visible outside the object. Instance variables are themselves objects with their own private data and public methods.

Methods are the code that is executed in response to a message. They are similar to normal colon definitions but use a special syntax using the words :M and ;M. You can put any Forth code inside a method including sending messages to other objects.

lnheritance allows you to define a class as a subclass of another class called the superclass. This new class "inherits" all of the instance variables and methods from the superclass. You can then add instance variables and methods to the new class. This can greatly decrease the amount of code you have to write if you design the class hierarchy carefully.

How to define a class

This example of a Point class illustrates the basic syntax used to define a class:

```
:Class Point <Super Object
    Var x
    Var y

    :M Get: ( -- x y ) Get: x Get: y ;M
    :M Put: ( x y -- ) Put: y Put: x ;M
    :M Print: ( -- ) Get: self SWAP ." x = " . ." y = " . ;M
    :M Classlnit: 1 Put: x 2 Put: y ;M
;Class
```

The class **Point** inherits from the class Object. Object is the root of all classes and defines some common behavior (such as getting the address of an object or getting its class) but does not have any instance variables. All classes must inherit from a superclass.

Next we define two instance variables, x and y. Both of these are instances of class Var. Var is a basic cell-sized class similar to a Forth variable. lt has methods Get: and Put: to fetch and store its data.

The Get: and Put: methods of class Point access its data as a pair of integers. They are implemented by sending Get: and Put: messages to the instance variables. Print: prints out the x and y coordinates.

Classlnit: is a special initialisation method. Whenever an object is created, the system sends it a Classlnit: message. This allows the object to perform any initialisation functions. Here we initialise the variables x and y to a preset value. Whenever a point is created, it will be initialised to these values. This is similar to a constructor in C++.

Not all classes need a Classlnit: method. lf a class does not define the Classlnit: method, there is one in class Object that does nothing.

Creating an instance of a class

Now we have defined the Point class, let's create a point:

```
Point myPoint
```

As you can see, Point is a defining word. It creates a Forth definition called myPoint. Let's see what it contains:

```
Print: myPoint
```

This should print the text "x = 1 y = 2" on the screen. You can see that the new point has been initialised with the ClassInit: message.

Now we can modify myPoint and we should see the new value:

```
3 4 Put: myPoint
Print: myPoint
```

Notice that in the definition of Point, we created two instance variables of class Var. The object defining words are "class smart" and will create instance variables if used inside a class and global objects if used outside of a class.

Sending a message to yourself

In the definition of Print: we used the phrase Get: self. Here we are sending the Get: message to ourselves. Self is a name that refers to the current object. The compiler will compile a call to Point's Get: method. Similarly, we could have defined ClassInit: like this:

```
:M Classlnit:   1   2   Put: self  ;M
```

This is a common factoring technique in Forth and is equally applicable here.

Creating a subclass

Let's say we wanted an object like myPoint, but one that printed itself in a different format.

```
Class  NewPoint   <Super Point
     :M  Print:  ( -- ) Get: self  SWAP  O .R  ." ©"  .  ;M
;Class
```

A subclass inherits all of the instance variables of its superclass, and can add new instance variables and methods of its own, or override methods defined in the superclass. Now let's try it out:

```
NewPoint    myNewPoint
Print:      myNewPoint
```

this will print `"1@2"` which is the Smalltalk way of printing points. We have changed the Print: method but have inherited all of the other behaviors of a Point.

Sending a message to your superclass

In some cases, we do not want to replace a method but just add something to it. Here's a class that always prints its value on a new line:

```
:Class  CrPoint  <Super  NewPoint
    :M  Print:  ( -- )  CR  Print:  super  ;M
;Class
CrPoint  myCrPoint
Print:   myCrPoint
```

When we use the phrase `Print: super` we are telling the compiler to send the print message that was defined in our superclass.

Windows messages and integers

When implementing winprocs and programming Windows, it is often useful to be able to treat the Windows message such as `WM_CHAR` as a method selector. The VFX Forth for Windows implementation allows any Windows message or integer to be used as a selector.

```
:M  WM _CHAR  ...  ;M
:M     $55AA  ...  ;M
```

A pseudo selector WM: has been defined that treats an item on the stack as a selector. Inside a winproc or other code, use WM: to pass the selector to the object.

```
<message/int> WM: object or
<message/int> WM: [ object ]
```

At a lower level, useful for passing Windows messages to an object without knowing the message until runtime, use MSG>OBJ which requires a selector and an object reference. If the method is applicable to the object, the appropriate method is executed and true is returned, otherwise false is returned.

```
<message> <object> MSG>OBJ
\ i*x message -object -- j*x true
\ i*x message -object -- i*x false
```

Indexed instance variables

Class Point had two named instance variables, "x" and "y." The type and number of named instance variables is fixed when the class is defined. Objects may also contain indexed instance variables. These are accessed via a zero-based index. Each object may define a different number of indexed index variables. The size of each variable is defined in the class header by the word <Indexed.

```
:Class Array <Super Object CELL <Indexed
:M At: ( index -- value ) (At) ;M
:M To: ( value index -- ) (To) ;M
;Class
```

We have declared that an Array will have indexed instance variables that are each CELL bytes wide. To define an array, put the number of elements before the class name:

```
10 Array myArray
```

This will define an Array with 10 elements, numbered from 0 to 9. We can access the array data with the At: and To: methods:

```
    4  At:  myArray  .
64  2  To:  myArray
```

Indexed instance variables allow the creation of arrays, lists and other collections.

Early vs. late binding

In these examples, you may have been thinking, "all of this message sending must be taking a lot of time." In order to execute a method, an object must look up the message in its class, and then its superclass, until it is found.

But if the class of the object is known at compile time, the compiler does the lookup then and compiles the execution token of the method. This is called "early binding." There is still some overhead with calling a method, but it is quite small. In all of the code we have seen so far, the compiler will do early binding.

There are cases when you do want the lookup to occur at runtime. This is called "late binding." An example of this is when you have a Forth variable that will contain a pointer to an object, yet the class of the object is not known until runtime. The syntax for this is:

```
VARIABLE objPtr myPoint objPtr !
Print: [ objPtr e ]
```

The expression within the brackets must produce an object address. The compiler recognized the brackets and will do the message lookup at runtime.

(Don't worry, "[" or "]" have not been redefined. When a message selector recognizes the left bracket, it uses PARSE and EVALUATE to compile the intermediate code and then compiles a late-bound message send. This also works in interpret state.)

Class binding

Class binding is an optimization that allows us to get the performance of early binding when we have object pointers or objects that are passed on the stack. If we use a selector with a class name, the compiler will early bind the method, assuming that an object of that class is on the stack. So if we write a word to print a point like this,

☛ ✝

: .Point (aPoint --) Print: Point ;

objPtr © .Point

✡ ✠

it will early bind the call. If you pass anything other that a Point, you will not get the expected result (It will print the first two cells of the object, no matter what they are). This is an optimization technique that should be used with care until a program is fully debugged.

Creating objects on the heap

If a system has dynamic memory allocation, the programmer may want to create objects on the heap at runtime. This may be the case, for instance, if the programmer does not know how many objects will be created by the application.

The syntax for creating an object on the heap is:

☛ ✝

Heap> Point objPtr !

✡ ✠

Heap> will return the address of the new point, which can be kept on the stack or stored in a variable. To release the point and free its memory, we use:

☛ ✝

objPtr © Release

✡ ✠

Before the memory is freed, the object will receive a Release: message. It can then do any cleanup necessary (like releasing other instance variables). This is similar to a C++ destructor.

Implementation

The address of the current object is stored in the user variable CURROBJ, and the contents are returned by -BASE. This means that the only time you can use -BASE is inside a method. Whenever a method is called, -BASE is saved and loaded with the address of the object being sent the message. When the method exits, -BASE is restored.

Class structure

All offsets and sizes are in Forth cells.

Offset	Size (cells)	Name	Description
0	8	MFA	Method dictionary (8-way hashed list)
8	1	!FA	Linked-list of instance variables
9	1	DFA	Data length of named instance variables
10	1	XFA	Width of indexed instance variables
11	1	SFA	Superclass pointer
12	1	TAG	Class tag field
13	User defined	User-defined field	

Neon-style Object Oriented Programming

The first 8 cells are an 8-way hashed list of methods. Three bits from the method selector are used to determine which list the method may be in. This cuts down search time for late-bound messages.

The IFA field is a linked list of named instance variables. The last two entries in this list are always "self" and "super."

The DFA field contains the length of the named instance variables for an object.

The XFA field actually serves a dual role. For classes with indexed instance variables it contains the width of each element. For non-indexed classes this field is usually zero. A special value of

-1 is a flag for general classes (see below).

The TAG field contains a special value that helps the compiler determine if a structure really represents a class.

The USR field is not used by the compiler but is reserved for a programmer's use. In the future I may extend this concept of "class variables" to allow adding to the class structure. This field is used in a Windows implementation to store a list of window messages the class will respond to.

Object structure

The first field of a global or heap-based object is a pointer to the object's class. This allows us to do late binding. Normally, the class field is not stored for an instance variable. This saves space and is not usually needed because the compiler knows the class of the instance variable and the instance variable is not visible outside the class definition. For indexed classes, the class pointer is always stored because the class contains information needed to locate the indexed data. Also, the programmer may mark a class as "general" so that the class pointer is always stored. This is needed in cases where the object sends itself late-bound messages (i.e. msg: [self]).

☛ ✝

Offset Size (cells) Description

0 1 Pointer to object's class

1 DFA Named instance variable data DFA+1 1 Number of indexed instance variables DFA+2 ? Indexed instance variables (if indexed)

✡ ✠

When an object executes, it returns the address of the first named instance variable. This is what we refer to when we mean the "object address." This field contains the named instance variable data. Since instance variables are themselves objects, this structure can be nested indefinitely.

Objects with indexed instance variables have two more fields. The indexed header contains the number of indexed instance variables. The width of the indexed variables is stored in the class structure, which is why we must always store a class pointer for indexed objects.

Following the indexed header is the indexed data. The size of this area is the product of the indexed width and the number of elements. There are primitives defined to access this data area.

Instance variable structure

The link field points to the next instance variable in the class. The head of this list is the IFA field in the class. When a new class is created, all the class fields are copied from the superclass and so the new class starts with all of the instance variables and methods from the superclass.

```
Offset (cells) Size (cells) Name Description
0    1   Link points to link of next ivar in chain
1    1   Name hash value of name
2    1   Class pointer to class
3    1   Offset offset in object to start of ivar data
4    1   #elem number of elements indexed ivars only)
```

The name field is a hash value computed from the name of the instance variable. This could be stored as a string with a space and compile-time penalty. But with a good 32-bit hash function collisions are not common. In any event, the compiler will abort if you use a name that collides with a previous name. You can rename your instance variable or improve the hash function.

Following the name is a pointer to the class of the instance variable. The compiler will always early-bind messages sent to instance variables.

The offset field contains the offset of this instance variable within the object. When sending a message to an object, this offset is added to the current object address.

If the instance variable is indexed, the number of elements is stored next. This field is not used for non-indexed classes.

Unlike objects, instance variables are not names in the Forth dictionary. Correspondingly, you cannot execute them to get their address. You can only send them messages. If you need an address, you can use the Addr: method defined in class Object.

Method structure

Methods are stored in an 8-way linked-list from the MFA field. A 32-bit selector identifies each method, which is the parameter field address of the message selector.

```
Offset (cells) Size (cells) Description
0    1   Link to next method
1    1   Selector
2    1   Method execution token
```

Methods are defined using the words :M and ;M which act like the familiar words : and ; exceptChapter 34: Neon-style Object Oriented Programming that they compile coded routines that manage the current object pointer as well as handling the procedure call. VFX Forth's use of coded routines makes the method call overhead negligible.

Selectors are special words

In the Yerk implementation, the interpreter was changed (by vectoring FIND) so that it automatically recognized words ending in ":" as a message to an object. It computed a hash value from the message name and used this as the selector. This kept the dictionary small.

In ANS Forth, there is no way to modify the interpreter (short of writing a new one). It has also been argued whether or not this is a "good thing" anyway.

In this implementation, message selectors are immediate Forth words. They are created automatically the first time they are used in a method definition. Since they are unique words, we use the parameter field of the word as the selector.

When the selector executes it compiles or executes code to send a message to the object that follows. If used inside a class, it first looks to see if the word is one of the named instance variables. If not, it sees if it is a valid object. Lastly it sees if it is a class name and does class binding.

Yerk also allowed sending messages to values and local variables and automatically compiled late-bound calls. In ANS Forth, we cannot tell anything about these words from their execution token, so this feature is not implemented. We can achieve the same effect by using explicit late binding:

```
Message: [ aValue ]
```

Object initialization

When an object is created, it must be initialised. The memory for the object is cleared to zero and the class pointer and indexed header are set up. Then each of the named instance variables is initialised.

This is done with the recursive word ITRAV. It takes the address of an instance variable structure and an offset and follows the chain, initializing each of the named instance variables in

the class and sending it a ClassInit: message. As it goes it recursively initializes that instance variable's instance variables, and so on.

Finally, the object is sent a ClassInit: message. This same process is followed when an object is created from the heap.

Example classes

Some simple classes have been implemented to serve as a basis for your own class library. These classes have similar names and methods to the predefined classes in Yerk and Mops. The code for the class implementation and sample classes is available from the Forth Interest Group (FIG) FTP site: ftp://ftp.forth.org/pub/Forth/ANS/CLASSOl.ZIP

http://www.forth.org

Conclusions

For me, the primary benefit of using objects is in managing complexity. Objects are little bundles of data that understand and act on messages sent by other parts of the program. By keeping the implementation details inside the object, it appears simpler to the rest of the program. Inheritance can help reduce the amount of code you have to write. If a mature class library is available, you can often find needed functionality already there.

If the Forth community could agree on an object-oriented model, we could begin to assemble an object-oriented Forth library similar to the Forth Scientific Library project headed by Skip Carter, code and tools that all Forth programmers can share. That project had not been possible before the ANS standardization

Chapter 7 Forth Internal Details

Internal Details, memory layout, stack locations,

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:

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
```

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 Local 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.

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.

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.

Chapter 8 Reference Material

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. It 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 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
```