kForth-32
User's Guide

Ver. 2.x

Table of Contents

Overview....5
Credits....6
1. Installation....7
1.1 Installation under GNU/Linux....7
1.1.1 Required Packages....7
1.1.2 kForth-32 on 64-bit Linux Systems....8
1.1.3 Library Packages for Forth Programming....8
1.2 Build and Configuration....9
2. Using kForth....11
2.1 Basics....11
2.2 More Words....11
2.3 Using Forth's Stacks....12
2.3.1 The Data Stack....13
2.4 Variables and Constants....14
2.5 Stack Diagrams....17
2.6 Simple Word Examples....18
2.7 Acting on Conditions....20
2.8 The Return Stack....22
2.9 Factoring a Forth Program....24
2.10 Using Memory....25
2.10.1 Data Types....25
2.10.2 CREATE and ALLOT....25
2.10.3 Viewing Memory with DUMP....27
3. Dictionary....29
3.1 Dictionary Maintenance....30
3.2 Word Lists and Search Order....30
3.3 Compilation and Execution Words....32
3.4 Defining Words....33
3.5 Control Structures....34
3.6 Stack Operations....35
3.7 Memory Operations....36
3.8 String Operations....38
3.9 Logic and Bit Manipulation Operations....38
3.10 Arithmetic and Relational Operations....39
3.10.1 Single and Double Integer Operations....39
3.11 Floating Point Operations....41
3.11.1 Arithmetic and Relational Words....41
3.11.2 Floating Point Functions....42
3.12 Number Conversion....42
3.13 Input and Output....43
3.14 File Access....45
3.15 Operating System Interface....47
3.16 Miscellaneous....48
4. Technical Information....49
4.1 Forth-94 Compliance....49
4.2 Threading Model....49
4.3 Signed Integer Division....50

4.4 Double Numbers....51
4.4.1 Double Number Entry....51
4.4.2 kForth Method....51
4.4.3 Forth-94/2012 Compatible Method....51
4.5 Floating Point Implementation....52
4.6 Special Features....52
4.7 Benchmarks and Tests....55
4.8 Exceptions and VM Error Codes....56
4.9 Source Code Map....57
4.10 Embedding kForth....57

Overview

kForth is a computer program that may be used in various ways:

1. It may be used as a calculator.
2. It may be used to run computer programs written in the Forth language.
3. It may be embedded into another computer program to give that program the ability to understand and run Forth programs.

kForth, in its simplest mode of use, can evaluate arithmetic expressions typed in by the user. Expressions are entered in a manner similar to that used for RPN (reverse Polish notation) calculators, such as for Hewlett-Packard scientific calculators. kForth permits arithmetic for single integer (32-bit), double integer (64-bit), and double-precision (64-bit) *floating point* numbers. It also provides built-in transcendental functions and other common number operations. Logic and bit operations may be performed, and the number base may be changed – numbers may be entered and displayed in hexadecimal (base 16), binary (base 2), or another base.

kForth is an implementation of the Forth programming language and environment. The user may write Forth programs with an editor, load these program files from kForth, and run them. kForth, like other implementations of Forth, provides an interactive environment, allowing the user to examine and define and execute individual *words*. Interactive use is one of the main advantages in using a Forth environment for writing and testing computer programs.

kForth-32, version 2.x provides a large *subset* of, and conforms primarily to the Forth-94 standard (ANS Forth) for the Forth language. It also includes new concepts and words from the Forth-2012 specification, and is intended to be transitional to a future kForth-32 3.x series which is more closely in conformance with Forth-2012. kForth also provides some extensions and non-standard features which its authors have found to be useful. Experienced Forth users should consult the Technical Information section of this User's Guide for specific information on the differences between kForth and Forth-94/Forth-2012.

Some notable features of kForth are:

- It is reasonably fast for many applications.
- It detects and reports many kinds of programming mistakes, providing useful feedback to aid the user in correcting his/her Forth program.
- It includes this User's Guide containing a beginner's tutorial on using Forth, describes the function of each of kForth's intrinsic words, and provides technical details about kForth for intermediate and advanced users.
- It comes with a large collection of example Forth programs, many of which are complete and useful programs.
- It provides a set of Forth source libraries for productive programming, including:
    - String manipulation, standard file access, and console output control

    - Structures, lists, simple objects, and a portable modular programming framework
    - A tested, precision numerical computing library, comprised of modules from the Forth Scientific Library with many extra modules, and scientific computing examples.
    - Operating system calls, sockets, signals, and communication with device drivers
    - Assembler for x86 processors

- Its shared library interface supports bindings to pre-compiled external libraries of functions written in C and Fortran. Library bindings are provided for X-Windows programming (`libX11`), GNU Multiple Precision Arithmetic Library (`libgmp`), and the GNU Multi-precision Floating Point Library (`libmpfr`) .
- It provides a low-level operating system interface for Linux, making it possible to write Forth programs for instrument control and data acquisition. Examples include communication via RS-232 serial and IEEE 488.2 (GPIB) interfaces.
- It simplifies using the large amounts of memory available to the computer, through its *dynamic dictionary* design.
- It provides a large amount of test code, written in Forth, to validate its own operation. Tests for compliance to Forth-94/Forth-2012 specified behavior and validation of its floating point arithmetic are among the provided system tests.

In addition to being as a stand-alone computing environment, kForth was designed to be easily embedded into another program. Advanced programmers, typically programming in the C and C++ languages, can use the kForth source code to make their own programs *user extensible*. In fact kForth was originally developed to allow users of XYPLOT for Linux to customize and add their own functions to the program. They can do this without modifying the XYPLOT program itself. Instead, they may write separate Forth modules and load them to extend XYPLOT's capabilities.

Credits

kForth was developed over several decades by its principal author, Krishna Myneni, with programming and technical contributions by the following people: David P. Wallace, Matthias Urlichs, Guido Draheim, Brad Knotwell, Alaric B. Snell, Todd Nathan, Bdale Garbee, Christopher M. Bannon, and David N. Williams. Others have graciously permitted porting of their work to kForth. If I have inadvertently omitted mention of anyone who has made technical contributions to kForth, please let me know at krishna.myneni@ccreweb.org.

1. Installation

kForth-32 is provided under the terms of the GNU Affero General Public License (AGPL). New releases of this software will be posted at GitHub as they become available. This manual provides a guide to the use and documents the features of kForth.

The source package is distributed as compressed tar (Unix Tape Archive) files:

- **`kForth-32-x.y.z.tar.gz`** (Linux/x86 version)

where `x.y.z` is the current version number, such as `2.1.5`. The source package unpacks to a directory of source files and a `Makefile` for building the executable(s). Difficulties with installation should be reported to: krishna.myneni@ccreweb.org

1.1 Installation under GNU/Linux

1.1.1 Required Packages

The following *packages* are required to build and maintain kForth-32 from its source package, on a GNU/Linux system:

- **`binutils`**
- **`gcc`**
- **`gcc-c++`**
- **`glibc`**
- **`glibc-devel`**
- **`libstdc++-devel`**
- **`make`**
- **`readline`**
- **`readline-devel`**
- **`patchutils`**

Note that some of the package names may be slightly different, depending on your GNU/Linux distribution. Some or all of these packages may already be installed on your GNU/Linux system, but if they are not, you should be able to install them manually for your distribution. You may use your system's graphical package manager to check for installation of the required packages, or use a command line query. For example, if your GNU/Linux system is `rpm`-based, you may verify that these packages have been installed by using the `rpm` command in the following way:

`rpm -q` *`package-name`*

The above command will return the version number of the package if it has been installed. The version of GNU C/C++ should be **`3.2`** or higher. On a Debian package-based system, the following command line query may be used:

aptitude search *package-name*

While it may seem tedious to determine the necessary package names and install any needed packages on your system, this is a one-time procedure which will enable your system to be used for building future releases from source code, and for software development.

1.1.2 kForth-32 on 64-bit Linux Systems

kForth-32 is always built as a 32-bit application, even on 64-bit systems. If you are building on a 64-bit system (`x86_64`), the 32-bit versions of of the C/C++ libraries and other libraries (`ncurses`, `readline`) must be installed. On a system such as CentOS 7, and other Red Hat Enterprise Linux 7 derived systems, additional packages are installed using

sudo yum install *package-name*

Installing the following additional packages will provide the needed libraries to build kForth on these systems:

- **glibc-devel.i686**
- **libstdc++-devel.i686**
- **ncurses-devel.i686**
- **readline-devel.i686**

1.1.3 Library Packages for Forth Programming

In addition to the packages needed to build kForth, additional libraries may be installed to allow Forth programs access to X11 graphics and multi-precision arithmetic. Various examples of Forth programs which use external libraries are provided in `forth-src/libs` and `forth-src/x11`. The following packages may be installed to run these examples (both 64-bit and 32-bit versions of the packages may coexist on a system).

- **libX11.i686**
- **libXft.i686**
- **libXrender.i686**
- **xorg-x11-fonts-75dpi.noarch**
- **xorg-x11-fonts-100dpi.noarch**
- **xorg-x11-fonts-Type1.noarch**
- **xorg-x11-fonts-misc.noarch**
- **xorg-x11-fonts-ISO8859-1-75dpi.noarch**
- **gmp.i686**
- **mpfr.i686**

1.2 Build and Configuration

Assuming your Linux system has the required packages, follow these steps to unpack, build, and install kForth:

1. Create a directory for the kForth source files, typically in your home directory, *e.g.*

   `mkdir ~/kforth`

2. Move the kForth archive file into this directory:

   `mv kForth-32-x.y.z.tar.gz ~/kforth`

3. Change to the `~/kforth` directory and extract the files:

   `cd ~/kforth`

   `tar -zxvf kForth-32-x.y.z.tar.gz`

   After this step, a subdirectory will be created with the name **`kForth-32-x.y.z`**. This directory will contain all of the kForth source files, the **`Makefile`**(s), as well as a **`README`** file with these same instructions.

4. Change to the `kForth-32-x.y.z` directory:

   `cd kForth-32-x.y.z`

5. Build the kForth executable. There are several options for building kForth, but the simplest is to type:

   `make`

   All of the source files will be compiled/assembled and two executable files, named `kforth32` and `kforth32-fast`, will be generated.

6. At this point you should be able to run the executables from your `~/kforth/kforth-x.y.z` directory. If you wish to make kforth available to all users or to place the programs in the default search path, move the executables to a suitable directory (`/usr/local/bin/` is recommended) using:

   `sudo mv kforth32 /usr/local/bin/`
   `sudo mv kforth32-fast /usr/local/bin/`

Any user should then be able to execute `kforth32` or `kforth32-fast`. You must have superuser privilege to do this last step.

7. Sample source code files are included in the archive. These files have extension `.4th`. Users may copy the example programs to their own directories.

8. You may specify a default directory in which kforth will search for `.4th` files not found in the current directory. The environment variable `KFORTH_DIR` must be set to this directory. For example, under the BASH shell, if you want the default directory to be `~/kforth/kForth-32-x.y.z/forth-src`, add the following lines to your `.bash_profile` file:

```
KFORTH_DIR=~/kforth/kForth-32-x.y.z/forth-src
export KFORTH_DIR
```

9. The file `kforth.xpm` may be used to create a desktop icon for kForth under X Windows. For example, if you are using the KDE environment, copy `kforth.xpm` to the `/usr/share/icons` directory.

2. Using kForth

2.1 Basics

Type **kforth32** to start the program. Upon startup, kForth will inform you that it is ready to accept input by displaying

```
Ready!
```

You may type commands, a sequence of *words*, and press `Enter`. kForth will respond with the prompt

```
ok
```

after it finishes executing each line of input. To illustrate, try typing the following

```
2 5 + .
```

and press Enter. kForth will respond with

```
7 ok
```

You may now enter another sequence of words. One particularly useful word to know is

```
bye
```

kForth will respond by saying

```
Goodbye
```

and exiting. kForth is not *case sensitive* – you may enter words in lower case *or* upper case.

2.2 More Words

The word

```
words
```

will display a list of currently defined words in the *dictionary*. You may define your own words by typing them at the kForth prompt. For example, a word that counts from one to ten and displays

each number counted may be defined by entering

```
: count_to_ten 10 0 do i 1+ . loop ;
```

The symbols "`:`" and "`;`" are very important – they indicate to the kForth compiler the beginning and ending of the definition of the word, called **`count_to_ten`** in this example. kForth will display the prompt `ok` after the new word has been compiled into the dictionary.

You can verify that our newly defined word has been added to the dictionary by using `words`. Now, execute the word by typing

```
count_to_ten
```

and pressing `Enter`. kForth will display the output

```
1 2 3 4 5 6 7 8 9 10 ok
```

If you are entering a definition that requires several lines of typing, the `ok` prompt will not be displayed until the end of the definition has been entered, *i.e.* until the compiler encounters a semicolon.

Although you can write Forth programs this way, it is much easier to create the definitions in a separate source file and then load them into kForth by issuing the command

`include` *`filename`*

For example, the definition of **`count_to_ten`** could have been entered into a plain text file called `prog1.4th`. Once kForth has been started, you can simply issue the command

```
include prog1
```

kForth will read the input from the specified file as though it was being entered from the keyboard. You may have noticed that the full file name was not entered in the **`include`** command. If no extension is specified, the file is assumed to have an extension of **`.4th`**.

You may also load a source file upon startup of kForth by typing

`kforth32` *`filename`*

2.3 Using Forth's Stacks

Forth provides reserved memory regions, called *stacks*, in which certain types of data may be placed and operated upon by defined words. One of these stacks is the data stack, often referred to

as just the “stack”. Another is called the *return stack*. We will discuss use of the data stack for performing computations on integer numbers and on *floating point* numbers. The return stack will be discussed in a later section.

2.3.1 The Data Stack

You may enter numbers onto the *data stack* simply by typing them and pressing Enter. You can use the word **`.S`** to list the contents of the stack. For example, type the following and press Enter.

```
2 5
```

You have placed two numbers onto the “stack”. Now, type

```
.S
```

and press `Enter`. kForth will respond by listing the items on the stack:

```
5
2
```

Notice that 5 is on the top of the stack – items are placed into the stack in a *first-in, last-out* order. Stack operators (words) are a part of the Forth language. Examples include the arithmetic operators

```
+ - * /
```

These operate on the top two items on the stack and replace them with the result. Other words change the order of items on the stack or copy or remove items from the stack:

```
SWAP ROT DUP OVER TUCK DROP NIP
```

Each data stack *cell* holds a single integer number.

You may also place representations of *real numbers*, called *floating point* numbers, onto the data stack. These numbers must be input in a special way known as *exponential notation*, for Forth’s interpreter to recognize them as floating point numbers. For example, to place the computer representation of the real number 3.14 onto the stack, type

```
3.14e0
```

and press `Enter`. The zero following the **`'e'`** indicates the power of ten that is multiplied to the number (10 raised to the zero power is equal to 1). Therefore, **`3.14e0`** corresponds to the real number, 3.14×10^0. If the exponent is zero, as in this example, the entry can be shortened to simply

```
3.14e
```

Exponential format allows you to enter very small and very large numbers easily. To enter the fractional number representing one-billionth, 0.000000001, or 1×10^{-9}, you may type

```
1e-9
```

and press `Enter`.

When you place a floating point number onto the stack and list the stack using **`.S`**, you will see two integer numbers printed instead of one real number. *A floating point number occupies two stack cells instead of one*, and **`.S`** lists the contents of each cell as though it were a single integer. You may print the floating point number occupying the top two cells of the stack with the word

```
F.
```

Use the words

```
F+ F- F* F/
```

to perform arithmetic on floating point numbers which have been placed onto the stack. For example,

```
3.14e 6.28e f+ f.
```

will print the result `9.42`. Words to manipulate floating point numbers on the stack include

```
FSWAP FROT FDUP FOVER FDROP
```

Computer users should be aware that floating point representations of real numbers are rarely *exact representations*, and arithmetic with floating point numbers will likely produce errors from the ideal mathematical result with real numbers. The *precision* with which real numbers are represented by floating point numbers affects the numerical *accuracy* of a floating point calculation on the computer. In kForth, the *default precision* for floating point numbers is that given by the IEEE 754 double-precision representation, which provides about 16 significant decimal digits for representing a real number.

2.4 Variables and Constants

An integer variable may be declared as follows:

variable *name*

Values may be stored and retrieved from the variable using the “store” (**!**) and “fetch” (**@**) operators. For example, if we want to define a variable called `counter` and initialize its value to 20, we enter the following:

```
variable counter
20 counter !
```

When you define a variable, memory is reserved at some *address* to hold an integer value, and the name of the variable becomes part of the dictionary. Typing the name **counter** at the Forth prompt and pressing enter will cause the memory address of **counter** to be placed onto the stack. Try the following:

```
counter
.s
```

You will see a memory address on top of the stack.

To examine the value stored in the variable `counter`, we place the address of counter on the stack, then use the fetch operator to retrieve the value from that address onto the stack, as follows.

```
counter @
```

The number 20 will be on top of the stack. Of course to see the value, we must print it using the word "dot" (**.**), so entering

```
counter @ .
```

will print the value 20. Forth also has a built-in word, **?**, that performs the sequence '**@ .** '.

Now, let's say we want to increment the value of **counter** by ten. First we fetch the value stored in **counter** onto the stack, then add ten, and finally store the new value into the variable. This is accomplished by the sequence,

```
counter @ 10 + counter !
```

Actually, Forth provides a shorter way of doing the same thing:

```
10 counter +!
```

Floating point variables are defined in a similar way:

fvariable *name*

The corresponding operators for storing and retrieving floating point numbers into the variable are **f!** and **f@**. Let's define a floating point variable called **velocity** and initialize it to zero.

```
fvariable velocity
0e velocity f!
```

Note that a floating point value of zero is entered as **0e** and we used the operator **f!** to store the value into **velocity**. If we now want to increment the value of **velocity** by 9.8, we can enter

```
velocity f@ 9.8e f+ velocity f!
```

kForth does not have a word called **f+!**, but with a little more familiarity with Forth, you may easily define such a word! To print a floating point value on the stack, use the word **f.** as explained previously. For example,

```
velocity f@ f.
```

will print the value 9.8.

Integer constants are defined as follows

value **constant** *name*

To define a constant called `megabyte`, for example, type

```
1048576 constant megabyte
```

I often can't remember how many bytes there are in a megabyte, so I may have written instead

```
1024 1024 * constant megabyte
```

Now, type the name of the constant and print the top item on the stack

```
megabyte .
```

and you will see printed the value `1048576`. Typing the name of the constant retrieves *its value* (not an address) onto the stack.

Floating point constants are defined in a similar fashion

fvalue **fconstant** *name*

To define a constant containing the acceleration due to gravity, 9.8 meters per second squared, type

```
9.8e fconstant g
```

The name of the constant is **g**. Typing

```
g f.
```

will print 9.8. Now, let's add the value of **g** to the value of **velocity** and print the result to illustrate the use of floating point variables and constants

```
velocity f@ g f+ f.
```

2.5 Stack Diagrams

The kForth dictionary contains many words that you may execute simply by typing them at the ok prompt. Some words expect values to have been placed on the stack when they begin executing. During execution of the word, these values may be removed from the stack and other values may be placed onto the stack. The values that are expected on the stack at the beginning of execution and those values that are returned on the stack at the end of execution are stated in the form of a *stack diagram* for the word. For example, the stack diagram for the word **NEGATE** is written as follows:

(n -- m)

This diagram indicates that a single integer *n* must be on the stack prior to executing **NEGATE**. After **NEGATE** finishes executing, the original value *n* has been removed from the stack and is replaced by a new single integer *m*. Try typing

```
3 negate
```

Now, list the items on the stack using **.S**.

A stack diagram is simply a *comment* which allows the programmer to understand the expectations for the stack(s) before and after a word is executed. Their presence is ignored by the Forth interpreter. Words that do not expect any items to be on the stack, and which do not return anything on the stack (*e.g.* **CR** and **DECIMAL**) have a stack diagram that looks like

(--)

The word **@** has the stack diagram

(a -- n)

with the meaning that **@** expects an address *a* on the stack and returns a single integer *n* on the stack. In contrast the word **!** has the stack diagram

```
( n a -- )
```

with the meaning that **!** expects two items to be on the stack, a single integer *n* and an address *a*, with "*a*" being the *top* item on the stack. During execution, both *n* and *a* are removed from the stack, the word **!** using and dispensing with them. Nothing is returned on the stack.

2.6 Simple Word Examples

Now let us practice writing some simple and useful words.

Example 1: Compounding Interest

Suppose we invest $1000 and we expect that it will grow with a yearly interest of 6%, which is compounded annually. What will be the final amount after 10 years?

We can determine the amount of interest accumulated after each year by taking 6% of the current amount and adding that to the current amount. For example, you can type the following to compute and print the amount at the end of the first year:

```
1000 dup 6 * 100 / + .
```

We placed the starting amount on the stack, then **dup**licated this value on the stack to compute 6% interest. Finally we add the top two numbers on the stack, the starting amount and the interest, and print the sum. If you are confused by the above example, it will help to print the contents of the stack using **.S** after you enter each word on a separate line,

```
1000 .S
dup .S
6 .S
* .S
100 .S
/ .S
+ .S
.
```

To solve the problem for 10 years, we simply need to repeat this calculation ten times. However, we must skip the first word, **1000** and the last word, **.**, in between years so that we can use the compounded amount from one year as the starting amount for the next year. The final result may be printed at the end.

Performing a repetetive calculation is easy in Forth – it is done with a **DO...LOOP**. The word **DO** expects two numbers on the stack. The difference between the two numbers is the number of times

that the words between **DO** and **LOOP** will be executed. The smaller number should be on top of the stack The following word illustrates using the **DO...LOOP** to solve this problem:

```
\  compound 6% interest on $1000 for 10 years and print answer

: compound10 ( -- )
    1000                \ starting amount
    10 0 do             \ do this for ten years
      dup 6 * 100 /     \ compute 6% interest of current amount
      +                 \ add interest to current amount
    loop                \ loop to next year
    .                   \ finally print the result
;
```

Executing the word **compound10** will display the answer `1786.`

Now let's generalize our word so that it is more useful. We want to be able to specify the starting amount, the interest, and the number of years to compound the interest. Finally, we want to print the result as before. The following word takes inputs from the stack, computes the final amount, and prints the answer:

```
: compound ( nstart npercent nyears -- )
     0 do                \ do this for nyears
       2dup * 100 /      \ compute interest on current amount
       rot +             \ add interest to current amount
       swap              \ swap items on stack to keep same order
     loop                \ loop to next year
     drop .              \ drop interest and print final amount
;
```

The word **compound** assumes that we have entered the starting amount, the percent interest per year, and the number of years onto the stack, as indicated in its stack diagram. Therefore, to solve the problem of our previous example using the more general word we would type

```
1000 6 10 compound
```

and press `Enter`. The same answer found previously will be displayed. But with our new word we can also determine the compounded growth after any number of years (except zero), at any interest rate, and for any starting amount. To see what our investment will grow to after 20 years, type:

```
1000 6 20 compound
```

To conclude this example, let's modify the word **compound** so that it prints a table of the accumulated amount at the end of each year:

```
: compound ( nstart npercent nyears -- )
     0 do
       2dup * 100 /
       rot +

       i 1+ 2 .r      \ print year right-justified in 2 character field
       9 emit         \ print a tab
       dup 6 .r   \ print year-ending amount right justified in 6 char field
       cr         \ advance to the next line
```

```
    swap
  loop
  2drop ;
```

Notice that we made use of the word **I** in the above example. **I** gets the *loop index* and places it on the stack. The loop index starts at the number on top of the stack when **DO** executes, which is 0 in this example. The loop index increments by one after each **LOOP**. You can look up in the dictionary other words that may not be familiar to you in this example, such as **1+**, **.R**, **EMIT**, and **CR**.

Finally, it is easy in kForth to send the output from the last example to a file instead of printing it on the screen. This is done by typing

```
>file interest.txt
1000 6 20 compound
console
```

The word **>FILE** redirects output from the screen (console) to the file name specified subsequently, `interest.txt` in the above example. The word **CONSOLE** closes the file and redirects output back to the screen. We used **>FILE** and **CONSOLE** to send the results of our interest calculations to a file, which can then be imported into a spreadsheet to make a chart!

2.7 Acting on Conditions

Nearly all computer programs, except for the simplest, will check to see if a specified condition is either true or false, and carry out different instructions based on the result. We have already seen how a **DO ... LOOP** works in Forth. In this special case, the word **LOOP** adds one to the loop counter and then checks whether or not the condition that the loop counter is equal to the ending count of the loop is true or false. Often, we will want to instruct the computer to check conditions that are not related to loops and then execute one sequence of words if the condition is true, or another sequence of words if the condition is false. Let's see how we can do this in Forth.

To start with, let's look at how to test a condition and how the result of the test is represented. As an example, our condition to be tested is whether or not the variable **X** is greater than 2. In Forth, such a test would be written as

```
X @ 2 >
```

We fetch the value of **X** onto the stack, next place the integer 2 on the stack, and then use the word **>** to check whether or not the number buried one cell deep into the stack is greater than the number on the top of the stack. The stack diagram for **>** is

```
n1 n2 -- b
```

Therefore, **>** removes both numbers from the stack and leaves a *boolean flag*, written as "*b*" in the stack diagram above. The flag *b* is itself another number, but it is a number that is always either **0** or **-1**. The value of the flag represents one of two states: *true*, corresponding to the value **-1** and *false*, corresponding to the value **0**. For convenience, Forth provides two predefined constants **TRUE** and **FALSE**. Try the following.

```
TRUE .

FALSE .
```

Now we have learned that the result of a test is a flag value, either *true* or *false*, placed on top of the stack. Although our example used the word **>**, other words in Forth can test for *equality* of two numbers, a *less than* condition, and perform several other comparisons.

A flag on top of the stack is used by the word **IF** to cause the computer to jump to different locations inside the executing word, based on the flag's value. This process is called *conditional branching* and all programming languages provide a way to do this. The word **IF** is part of a *control structure* made up of the words **IF ... ELSE ... THEN**, where **...** represents some arbitrary word sequences. Many other programming languages have a structure similar to this, but in Forth its use is slightly different. The word **IF** assumes the conditional test has already been performed and that there is a flag on top of the stack. Let's illustrate the use of the **IF ... ELSE ... THEN** structure with an example. Suppose we want to write a word that prints whether a number given to it is “even” or “odd”. We could define this word as follows

```
: parity  ( n -- | print whether a number is even or odd )
    2 MOD 0=
    IF
      ." even"
    ELSE
      ." odd"
    THEN ;
```

In our definition of the word **parity**, the conditional test is given by the line

```
2 MOD 0=
```

The word **MOD** performs a division, except that it returns the *remainder* instead of the quotient. An “even” number divided by 2 has a zero remainder, so we check to see if the value returned by **MOD**, on top of the stack, is equal to zero. The word **0=** returns a true flag when the number on top of the stack is zero, a false flag otherwise. When **IF** examines this flag, if it finds the flag to be true, execution jumps to the word following **IF**. On the other hand, if the flag is false, execution branches to the word following **ELSE**. To see how it works, try typing a number followed by the word **parity**, e.g.

```
4 parity
```

A few other points to note about the **IF ... ELSE ... THEN** structure:

- When the word **IF** examines the flag on top of the stack, it treats *any* non-zero value as representing *true*. A zero value always corresponds to *false*. Therefore, we could define the word **parity** as:

```
: parity ( n -- ) 2 MOD IF ." odd" ELSE ." even" THEN ;
```

Notice the exchange of **`." odd"`** and **`."even"`** in the new version of **parity**.

- In some cases we may not want to do anything when the condition is false. For example, suppose we want to write a word that prints “odd” only when the number we give it is odd, but does nothing if the number is even. Then we can omit the **`ELSE ...`** portion of the structure. For example, we can define

```
: odd? ( n -- ) 2 MOD IF ." odd" THEN ;
```

Try passing different numbers to the word **odd?**, such as

```
5 odd? .
```

- When the condition flag is *true*, the words enclosed between **IF** and **ELSE** are executed; when the flag is *false*, the words enclosed between **ELSE** and **THEN** are executed. After either branch is executed, the computer resumes execution after the word **THEN**. The two branches come back together again following **THEN** – this is a feature of *structured programming*, which makes it easier for a person to trace the possible paths a computer may take through a sequence of instructions.
- An **`IF ... ELSE ... THEN`** structure can be placed inside a branch of another **`IF ... ELSE ... THEN`** structure. This is called *nesting*, and you will see an example of nested structures in the next section.

2.8 The Return Stack

Because Forth words often use values on the stack for input data, it may become difficult to keep the items ordered exactly as needed during the calculation, especially when there are several input values required. While Forth provides several stack manipulation words such as **`DUP SWAP ROT`**, etc., sometimes the most convenient operation is to temporarily remove an item from the top of the stack, then place it back on the stack when needed. Of course we may use variables for this purpose, but Forth provides a simpler way to accomplish this by providing another stack, called the *return stack*. An item on the stack can be temporarily “pushed” onto the return stack by using the word **`>R`**. The item may be “popped” back from the return stack to the ordinary data stack with the word **`R>`**. The return stack’s main purpose is to keep information on where to return after a word finishes executing; however, it may serve as temporary storage when carefully used.

The following example also illustrates the use of the return stack.

```
: this_date ( -- day month year )
     time&date >r >r >r 2drop drop r> r> r> ;
```

The word **this_date** returns today's date on the stack with the year on top. It does this by calling kForth's built-in word, **TIME&DATE**, which has the following stack diagram:

```
time&date ( -- secs mins hours day month year )
```

We want our word **this_date** to only return the *day*, *month*, and *year*, so we must remove *secs*, *mins*, and *hours* left on the stack by **TIME&DATE**. However, *day*, *month*, and *year* are on top and the three numbers we want to drop (*secs*, *mins*, and *hours*) are buried underneath. Using **>R** three times, we remove the *year*, *month*, and *day* from the stack, in that order. These numbers are pushed onto the return stack. Now we use **2DROP** and **DROP** to remove *hours*, *mins*, and *secs* from the stack. Finally, we use the word **R>** three times to pop the *day*, *month*, and *year* from the return stack back onto the data stack.

A word of caution to the novice Forth user: the return stack must be used with the following restrictions because Forth itself places items on the return stack at the beginning of executing a word and also when executing **DO** loops:

- Inside the definition of a word, every item pushed onto the return stack with **>R** must be popped from the return stack with a corresponding **R>** before the end of the word.
- There must also be a matching **R>** for every **>R** inside of a **DO ... LOOP**.
- Inside of **DO** loops, the loop index words **I** and **J** must not be used when items have been pushed onto the return stack but not yet popped.

Example 2: Calculating Age

In this example, we will make use of what we have learned up to now to compute the age of a person given their birth date. Following good Forth practice, we will first define a few simple words which we anticipate will be helpful for writing the actual age calculator:

```
: this_year ( -- year )
     this_date >r 2drop r> ;

: this_month ( -- month )
     this_date drop nip ;

: this_day ( -- day )
     this_date 2drop ;
```

The words **this_year**, **this_month**, and **this_day** all use **this_date**, defined previously, and remove any extra items from the stack. A couple more words will be helpful in our calculation:

```
: date< ( day1 month1 day2 month2 -- flag )
      rot swap
      2dup           \ is month1 less than month2?
      < if
        2drop 2drop  \ remove items on stack -- no further test needed
        true         \ leave true flag on the stack
      else
         = if        \ is month1 equal to month2?
           <         \ flag represents day1 less than day2
         else
```

```
        2drop       \ remove items on stack -- month1 > month2
        false       \   so return false flag
      then
    then ;
```

Notice that we used two nested **IF ... ELSE ... THEN** structures in our definition of **DATE<**. The first **IF** examines the flag returned by **<**, which tests whether or not *month1* is less than *month2*. If *month1* is not less than *month2*, we must then check to see if *month1* is equal to *month2*. The word **=** tests this condition and returns the appropriate flag, which is examined by the second **IF**.

```
\ test whether day and month are in future

: after_today ( day month -- b )
     this_day this_month 2swap date< ;
```

We are ready now to calculate a person's age, given their birth date.

```
: age ( day month year -- age | calculate age given birth date )
     this_year swap -   \ number of years between birth year and this year
     -rot               \ move top item to bottom of stack
     after_today if     \ is birthday later than today?
       1-               \ yes, subtract one from number of years
     then ;
```

We may test our definition of **AGE** by typing

day month year **age .**

where *day*, *month*, and *year* are the numbers for your birth day, month, and year. kForth will respond by printing your current age.

2.9 Factoring a Forth Program

You may have noticed that in our example of the age calculator, we defined several words, not just one. Breaking the calculation into individual short words is a way to make writing a program simpler, easier to understand, and easier to test when, as is inevitable, a program doesn't work like you imagined. Previously defined words can be used to write higher level words, making the higher level words more readable.

Well-written Forth programs will often have short low-level words, each of which performs a single and simple computation matching well the name of that word. As an example, consider the game `tetris.4th` written in Forth (see `forth-src/games/`). Notice how the words defined towards the beginning of the program, such as **DRAW-PIT** and **UPDATE-SCORE** are short words with well-defined functions matching their names. Near the end of the program, various words are combined to define the higher level word, **PLAY-GAME**. Although the working of the lower level words may not be immediately apparent from reading their definitions, in a properly *factored* Forth

program, the high level word(s), such as **PLAY-GAME**, are readable and often times resemble a natural language description of the actions performed by the word. Good factoring is a skill acquired through practice with writing programs in any programming language, and results in programs which are more easy to diagnose and repair when things don't work as expected.

2.10 Using Memory

2.10.1 Data Types

Earlier, in section 2.4, we learned how to create named integer and floating point variables using **VARIABLE** and **FVARIABLE**. These words *define* new words, which when executed, return the starting location (address) of the memory region containing their value. Different *data types* such as integers and floating point numbers require different amounts of memory for storing their values, and the words **VARIABLE** and **FVARIABLE** automatically reserve (**ALLOT**) the appropriate sized region.

In addition to the **VARIABLE** and **FVARIABLE** data types, Forth also provides **2VARIABLE** for a *double length* integer. Thus, **VARIABLE** will **allot** one *cell* (4 bytes on a 32-bit system) and **2VARIABLE** will **allot** two cells of memory. To find out how many bytes of memory represent one cell in a Forth system, type

```
1 cells .
```

We use **2VARIABLE**s, for example, when we want to store integer numbers that are too large, or will become too large in the course of executing our program, to be represented by *single length* **VARIABLE**s. The largest signed single length integer representable on a 32-bit Forth system can sometimes be limiting, necessitating the use of double length integers. On a 32-bit system, the range of signed whole numbers which can be represented by a single stack cell is

−2,147,483,648 to +2,147,483,647

For double length integers, the range of signed numbers which can be represented is

−9,223,372,036,854,775,808 to +9,223,372,036,854,775,807

Should you need to perform arithmetic which results in integers within that range, kForth provides double length integer arithmetic operators such as, **D+** and **D-**, as well as a number of mixed length arithmetic operators between single and double length integers.

2.10.2 CREATE and ALLOT

In writing our own Forth programs, we may need to store and retrieve data of different size than the sizes given by the data types discussed above. Examples are a paragraph of text, or an *array* of integers. How do we go about reserving memory for, say, 100 single length integers? In addition to reserving the memory, we need to assign a name with which to refer to the memory region. These tasks are accomplished through the use of the words **CREATE** and **ALLOT**:

```
CREATE iarray 100 CELLS ALLOT
```

The above statement will create a new word in the dictionary, called **iarray**, and reserve 100 cells (400 bytes on a 32-bit system). Executing the word **iarray** will return the starting address of the memory region. The words **CREATE** and **ALLOT** are, in fact, primitive Forth words which may be used to *define* words such as **VARIABLE**, *e.g.*

```
: VAR CREATE 1 CELLS ALLOT ;
```

Example 3: Initializing and Printing an Array of Integers

In our example above, we reserved a memory region of 100 cells in size using **ALLOT**. Simply **allot**ing this memory does not specify what is initally stored in this region. We might need to set the initial values of the 100 integers in **iarray** before using it in our computation. A word to set all of the 100 integers to zero could be defined in the following way.

```
: init-iarray ( -- | initialize iarray to zeros)
     iarray  100 0 DO  0 over ! cell+  LOOP  drop ;
```

Study the above example to see how the word performs the action of storing a zero in each of the 100 cells. You may look up the action of the word **CELL+** in the dictionary. A word to print the 100 integers stored in **iarray** may be defined as follows.

```
: print-iarray ( -- )
     cr  iarray
     100 0 DO
       dup @
       6 .R  i 1+ 8 mod 0= IF cr THEN   \ nice output formatting
       cell+
     LOOP  drop ;
```

Forth also provides the words, **FILL**, **BLANK**, and **ERASE**, to set all of the *bytes* in a memory region to a single byte value. Using **ERASE**, the word **init-iarray** may also be defined as

```
: init-iarray ( -- ) iarray 100 cells erase ;
```

Exercise: Try modifying our first definition of **init-iarray** so that it stores a running count from 1 to 100 in **iarray**, instead of initializing all the values to zero. The following output should be produced by **print-iarray**.

```
print-iarray


     1     2     3     4     5     6     7     8
     9    10    11    12    13    14    15    16
    17    18    19    20    21    22    23    24
    25    26    27    28    29    30    31    32
    33    34    35    36    37    38    39    40
    41    42    43    44    45    46    47    48
    49    50    51    52    53    54    55    56
    57    58    59    60    61    62    63    64
```

```
 65    66    67    68    69    70    71    72
 73    74    75    76    77    78    79    80
 81    82    83    84    85    86    87    88
 89    90    91    92    93    94    95    96
 97    98    99   100 ok
```

Exercise: Write a more general version of **`init-iarray`** which takes the array and the number of elements which may be stored in the array as arguments. The new word and its stack diagram will be,

`init-array` `( a n -- )`

where `a` is the starting memory address of the array and `n` is the number of elements in the array. Write a similarly generalized word to print any specified array or arbitrary length,

`print-array` `( a n - )`

2.10.3 Viewing Memory with DUMP

In some applications, particularly those involving sending and receiving data between the computer and another device, it is often very useful to be able to view the individual bytes stored in a region of memory. Forth provides the word **`DUMP`** to allow the user to view the individual byte contents of a memory region. In kForth, the word **`DUMP`** is provided as a *source definition*, that is, a word defined using more primitive Forth words, within the file `dump.4th`. To use the word **`DUMP`**, we must first **`include`** this file with

```
include dump
```

Then, typing **`IARRAY 64 DUMP`** should output something like

```
134969216 :  01  00  00  00  02  00  00  00  03  00  00  00  04  00  00  00  ................
134969232 :  05  00  00  00  06  00  00  00  07  00  00  00  08  00  00  00  ................
134969248 :  09  00  00  00  0A  00  00  00  0B  00  00  00  0C  00  00  00  ................
134969264 :  0D  00  00  00  0E  00  00  00  0F  00  00  00  10  00  00  00  ................ ok
```

At first glance, the above output does not seem too useful; however, if we look closely, the data stored previously in **`iarray`** may be seen – the running count starting from one can be seen in the successive groups of four bytes. Also, **`DUMP`** displays the individual bytes in base 16, or *hexadecimal*. This is not immediately apparent, until we see that the number 10 in **`iarray`** is displayed as the four-byte sequence "`0A 00 00 00`". Engineers trying to debug programs communicating with hardware often find “hex” output to be more useful than the ordinary decimal representation because it allows them to visualize the bit-pattern represented by each hex character.

`DUMP` also shows the address of the first byte of each line on the left hand side, and shows additional characters on the right hand side. When the bytes in memory represent printable characters, also known as ASCII codes, the corresponding character is displayed on the right hand side. To see this, try **`IARRAY 64 CELLS + 128 DUMP`**. The following output will be shown by **`DUMP`**:

```
134969472 :  41  00  00  00  42  00  00  00  43  00  00  00  44  00  00  00  A...B...C...D...
134969488 :  45  00  00  00  46  00  00  00  47  00  00  00  48  00  00  00  E...F...G...H...
134969504 :  49  00  00  00  4A  00  00  00  4B  00  00  00  4C  00  00  00  I...J...K...L...
```

```
134969520 :  4D  00  00  00  4E  00  00  00  4F  00  00  00  50  00  00  00  M...N...O...P...
134969536 :  51  00  00  00  52  00  00  00  53  00  00  00  54  00  00  00  Q...R...S...T...
134969552 :  55  00  00  00  56  00  00  00  57  00  00  00  58  00  00  00  U...V...W...X...
134969568 :  59  00  00  00  5A  00  00  00  5B  00  00  00  5C  00  00  00  Y...Z...[...\...
134969584 :  5D  00  00  00  5E  00  00  00  5F  00  00  00  60  00  00  00  ]...^..._...`... ok
```

3. Dictionary

1. Dictionary Maintenance
2. Word Lists and Search Order
3. Compilation and Execution Words
4. Defining Words
5. Control Structures
6. Stack Operations
7. Memory Operations
8. String Operations
9. Logic and Bit Manipulation Operations
10. Arithmetic and Relational Operations
11. Floating Point Words
12. Number Conversion
13. Input and Output
14. File Access
15. Operating System Interface
16. Miscellaneous

All of the words provided by kForth-64 are documented in this chapter. The notation used to express their stack diagrams is shown in the table below.

Arg Prefix	Data Type	Stack	Stack Cells
a	address	data	1
n	signed single integer	"	1
u	unsigned single integer	"	1
d	signed double length integer	"	2
ud	unsigned double length integer	"	2
t	signed triple length integer	"	3
ut	unsigned triple length integer	"	3
b	boolean flag: true or false (-1 or 0)	"	1
r	double precision floating point value	"	2
^str	counted string address	"	1
x	value of any single cell type	"	1
xt	execution token	"	1
nt	name token	"	1
wid	word list identifier	"	1

Word names which are **UNDERLINED** are either not part of the Forth-94 or Forth-2012 standards, or have usage with additional constraints beyond those specified in these standards. The few words which may have non-standard behavior in kForth rarely cause any difficulty for writing programs

which run on both kForth and other Forth-94/Forth-2012 compliant systems; however, the differences should be noted when attempting to run the programs on other systems.

Pleas refer to the Forth-94/Forth-2012 standards document for definitions of special terms. Examples are *execution token*, *name token*, *interpretation semantics*, and *compilation semantics*.

3.1 Dictionary Maintenance

FORGET	`--`	parse the next word in the input stream and remove the word and all subsequently defined words from the dictionary
COLD	`--`	restore the Forth environment to the startup state
WORDS	`--`	list the defined words in the current search order

The word **FORGET** may be used to remove words from the dictionary. Typing

FORGET *name*

will remove *name* and all words defined after *name* from the dictionary.

The word **COLD** deletes all non-intrinsic wordlists, definitions and strings, resets the search order and all stacks, and restarts the Forth environment in interpretation state – this is useful when you want to start over with the Forth system in a known state.

3.2 Word Lists and Search Order

Words in the dictionary are grouped into *word lists*. kForth provides the following built-in word lists:

Root Forth Assembler

New definitions are added to the current *compilation word list*, which is initially the `Forth` word list. When the compiler searches the dictionary for a word name, the search proceeds in a specified order through a specified series of word lists. This set of ordered word lists is known as the *search order*. The `Root` word list must always be a part of the search order; however, any other word lists may be added or removed from the search order. The user may create custom word lists to group new words added to the dictionary, and control their visibility to the Forth compiler.

ORDER	`--`	display the word lists in the search order. The word list at the beginning of the search order is displayed to the left, and the compilation word list is shown in brackets
GET-ORDER	`-- widn ... wid1 n`	return the word list address identifiers, and the number of word lists; `wid1` is the first

		word list in the search order.
SET-ORDER	widn ... wid1 n --	set the search order to the specified sequence of word lists, where wid1 is the first word list in the search order
ONLY	--	remove all word lists from the search order, except the minimal **Root** word list
FORTH	--	replace the first word list in the search order with the **Forth** word list
ASSEMBLER	--	replace the first word list in the search order with the **Assembler** word list.
GET-CURRENT	-- wid	return the word list address for the current compilation word list.
SET-CURRENT	wid --	set the compilation word list to be the word list identified by wid
FORTH-WORDLIST	-- wid	return the word list address identifying the **Forth** word list.
ALSO	--	duplicate the word list at the beginning of the search order.
PREVIOUS	--	remove the first word list in the search order
DEFINITIONS	--	set the first word list in the search order as the compilation word list.
WORDLIST	-- wid	create a new empty word list and return its identifier
VOCABULARY	--	create a new named word list which, when executed, will replace the first word list in the search order.
SEARCH-WORDLIST	a u wid -- 0 | xt n	search the word list identified by wid for the word name in the string, a u. Return n=0 if the word is not found in the word list, n=1 if the word is found and is an immediate word, n=-1 if the word is found and is not an immediate word.
TRAVERSE-WORDLIST	i*x xt wid -- j*x	Execute xt for every word in wordlist wid, passing each word's nt to xt, until the wordlist is exhausted or until xt returns false. See Forth 2012 spec. for additional information on TRAVERSE-WORDLIST.
FIND	^str -- xt n	search all the word lists in the search order for the word specified by the counted string; n is 0 if not found, n is 1 if found and the word is an IMMEDIATE word, n is -1 if found and the word is not an immediate word, and xt is a valid execution token if the word is found
[DEFINED]	-- b	parse a word name from the input stream,

		search for the name in the search order, and return a flag indicating whether or not the name was found (**TRUE** if found).
[UNDEFINED]	-- b	parse a word name from the input stream, search for the name in the search order, and return a flag indicating whether or not the name was found (**FALSE** if found).
NAME>STRING	nt -- a u	Return name string for word nt
NAME>COMPILE	nt -- x xt	Return compilation semantics of word nt
NAME>INTERPRET	nt -- xt | 0	Return interpretation semantics for word nt

3.3 Compilation and Execution Words

IMMEDIATE	--	set the precedence during compilation of the most recently defined word
NONDEFERRED	--	set the precedence during interpretation of the most recently defined word
POSTPONE	--	Parse the next word in the input stream and append its compilation semantics to the current definition
LITERAL	n | a --	compile a number or address from the stack into the current definition
2LITERAL	d --	compile a double number from the stack into the current definition
SLITERAL	a u --	compile a string address and count from the stack into the current definition
FLITERAL	r --	compile a floating point number from the stack into the current definition
'	-- xt	parse the next word in the input stream and return its execution token.
[']	--	immediate version of **'** for use inside a word definition
>BODY	xt -- a	convert the xt of a word to its parameter field address
COMPILE,	xt --	append the execution semantics, xt, to current definition
COMPILE-NAME	nt --	append execution semantics of word nt to current def.
EXECUTE	xt --	execute the semantics specified by xt
EXECUTE-BC	a -- i*x	execute kForth byte code starting at address a
EVALUATE	a u --	interpret and execute source code contained in a string
:	--	begin a new word definition and enter compilation state
;	-- | -- xt	Terminate a named or un-named definition and return to interpretation state. For un-named definition., return xt
:NONAME	--	begin an un-named definition and enter compilation state
[	--	enter interpretation state
]	--	enter compilation state
STATE	-- a	return address containing a flag: **true** if compiling, **false** if interpreting.

The words **'** (TICK), and **[']** behave according to the Forth-2012 standard, and return an execution token on the stack. The word **EXECUTE** may be used to execute a word given the execution token on the stack. The word **NONDEFERRED** is a non-standard word which is used to set the enhanced *precedence* state of a word in kForth. For more information on the concept of precedence in kForth, refer to the Technical Information section of the user's guide.

The following standard compilation words are provided in Forth source in `ans-words.4th`:

TO	`--`	determine the body address of the next word and append the run-time semantics to store a value at that address

3.4 Defining Words

In addition to ordinary “colon definitions” of the form,

: NAME ... ;

the following *defining words* are also provided:

CREATE *name*
VARIABLE *name*
n **CONSTANT** *name*
2VARIABLE *name*
d **2CONSTANT** *name*
FVARIABLE *name*
f **FCONSTANT** *name*

Both **:** and **CREATE** may be used inside a word definition to make your own defining words. The word **DOES>**, as part of a **CREATE ... DOES>** expression, allows you to specify the run time behavior of words created by the defining word.

The following common Forth defining words have source code definitions, provided in `ans-words.4th`:

n **VALUE** *name*
DEFER *name*

An existing word may be referred to by another name, using the standard word **SYNONYM**, defined in the Forth source file, `ans-words.4th`. Another way is to use the built-in non-standard word **ALIAS**.

SYNONYM	`--`	create a new word which has the same behavior as an existing word
ALIAS	`xt --`	create a new word which has the same execution behavior as `xt`

They may be used as follows,

```
SYNONYM name2 name1
' name1 ALIAS name2
```

where *name1* is the name of an existing word in the search order, and *name2* is the new name.

3.5 Control Structures

The following control structures are provided in kForth:

```
DO ... LOOP
DO ... +LOOP
?DO ... LOOP
?DO ... +LOOP
IF ... THEN
IF ... ELSE ... THEN
BEGIN ... AGAIN
BEGIN ... UNTIL
BEGIN ... WHILE ... REPEAT
CASE ... OF ... ENDOF ... ENDCASE
```

All control stuctures may be nested. For **DO** loops the number of levels of nesting is limited only by return stack space. The following execution control words are also defined:

```
RECURSE
LEAVE
EXIT
QUIT
ABORT
ABORT"
```

RECURSE compiles the execution semantics of the current definition *in* the current definition. It allows for recursive execution of a word. Note that other languages provide recursion by using a call to the same name as the function/procedure being described; however, this is not possible in standard Forth since the same name may exist within the search order. In standard Forth, the use of *name* within the definition of *name* is prescribed to refer to the prior definition of *name* within the search order. A classic example of using recursion is

```
: gcd ( n1 n2 -- gcd )  \ find greatest common divisor
    ?DUP IF SWAP OVER MOD RECURSE THEN ABS ;
```

LEAVE removes the current loop parameters from the return stack, by calling **UNLOOP**, and causes

an immediate jump out of the current loop. Execution resumes at the instruction following the loop.

EXIT returns from the word currently being executed. **EXIT** from within a loop requires that the loop parameters be discarded from the return stack explicitly with **UNLOOP**.

QUIT empties the return stack, terminates execution of the current word and returns kForth to the interpreter mode.

ABORT empties the data stack and executes **QUIT**.

ABORT" examines the flag on top of the stack and if the flag is true, prints the message delimited by ", then executes **ABORT**.

The exception handling words **CATCH** and **THROW** are defined in source in `ans-words.4th`.

3.6 Stack Operations

DUP	`x -- x x`	duplicate
?DUP	`x -- x x | 0`	dup if not zero
SWAP	`x1 x2 -- x2 x1`	swap
OVER	`x1 x2 -- x1 x2 x1`	over
ROT	`x1 x2 x3 -- x2 x3 x1`	rotate cw
-ROT	`x1 x2 x3 -- x3 x1 x2`	rotate ccw
DROP	`x --`	drop
NIP	`x1 x2 -- x2`	nip
TUCK	`x1 x2 -- x2 x1 x2`	tuck
PICK	`... n -- ... xn`	copy nth item deep
ROLL	`... n -- ... xn`	rotate nth item deep to top of stack
DEPTH	`... -- ... u`	data stack depth
2DUP	`x1 x2 -- x1 x2 x1 x2`	
2SWAP	`x1 x2 x3 x4 -- x3 x4 x1 x2`	
2OVER	`x1 x2 x3 x4 -- x1 x2 x3 x4 x1 x2`	
2ROT	`x1 x2 x3 x4 x5 x6 -- x3 x4 x5 x6 x1 x2`	
2DROP	`x1 x2 --`	
FDUP	`r -- r r`	duplicate a floating point number on top of the stack. Note: `r` occupies two stack cells
FSWAP	`r1 r2 -- r2 r1`	swap two floating point numbers on the stack
FOVER	`r1 r2 -- r1 r2 r1`	copy the floating point number one deep onto top of stack

FROT	r1 r2 r3 -- r2 r3 r1	rotate the order of three fp numbers on the stack
FDROP	r --	drop a floating point number from the stack
F2DROP	r1 r2 --	drop two fp numbers from the stack
F2DUP	r1 r2 -- r1 r2 r1 r2	duplicate a pair of fp numbers on the stack

Return stack operations are:

>R	x -- R: -- x	push x onto return stack
R>	-- x R: x --	pop x from return stack
R@	-- x R: x -- x	copy x from top of return stack
2>R	d -- R: -- d	push two stack cells onto return stack
2R>	-- d R: d --	pop two cells from return stack
2R@	-- d R: d -- d	copy two cells from top of return stack
I	-- n	current loop index
J	-- n	next outer loop index
UNLOOP	--	discard loop parameters from return stack

Note that **2>R** is *not equivalent* to the sequence **>R >R**. The order of the two single length elements on top of the return stack is different for the two cases. **2>R** pushes two items from the top of the stack so that they have the same order on the return stack. The sequence **2>R 2R>**, however, is identical to the sequence **>R >R R> R>**.

3.7 Memory Operations

@	a -- n	fetch single
!	n a --	store single n to address a
2@	a -- d	fetch double number from address a
2!	d a --	store double number to address a
A@	a1 -- a2	fetch address from address a
C@	a -- n	fetch byte
C!	n a --	store byte
W@	a -- n	fetch signed word (this word is obsolete – use SW@)
SW@	a -- n	fetch signed word
UW@	a -- u	fetch unsigned word
W!	n a --	store signed word
SL@	a -- n	fetch signed dword (same as @ on 32-bit system)
UL@	a -- u	fetch unsigned dword (same as @ on 32-bit system)
L!	n a --	store signed/unsigned dword (same as ! on 32-bit system)

SF@	a -- r	fetch single precision float
SF!	r a --	store r as single precision float
DF@	a -- r	fetch double precision float
DF!	r a --	store double precision float
F@	a -- r	same as **DF@**
F!	r a --	same as **DF!**
SP@	-- a	fetch data stack pointer
RP@	-- a	fetch return stack pointer
SP!	a --	set data stack pointer
RP!	a --	set return stack pointer
?	a --	fetch and print single; equivalent to **@ .**
ALLOT	u --	allocates u bytes in the dictionary
ALLOT?	u -- a	allocates u bytes in the dictionary and returns starting address of the allocated region
ALLOCATE	u -- a n	reserve u bytes of system memory and return starting address of the allocated region and error code
FREE	a -- n	release memory previously reserved with **ALLOCATE** and return error code (0 = success)
RESIZE	a1 u -- a2 ior	change size of previously **ALLOCATE**d region to u bytes; ior = 0 if success
C"	-- ^str	compile a counted string into the string table; the string is parsed from the input stream and must be terminated by **"**
S"	-- a u	compile a string and return address and count
COUNT	^str -- a u	convert counted string address to character buffer address a and character count u
MOVE	a1 a2 u --	move u bytes from source a1 to dest a2 ; handle overlapping region
FILL	a u1 n2 --	fill u1 bytes with byte value n2 starting at a
ERASE	a u --	fill u bytes with zero starting at a

The following standard memory words are provided in Forth source in `ans-words.4th` and `dump.4th`:

PAD	-- a	return address of a scratch-pad in memory for temporary use	ans-words.4th
DUMP	a u --	output a hexadecimal display of u bytes starting at address a	dump.4th

The non-ANS standard word **A@** is needed because kForth performs type checking for operands involved in memory access. **A@** performs **@** and sets the type field in the hidden type stack to represent an *address* for the retrieved value. Addresses may be stored in ordinary variables using **!**; however they should be retrieved with **A@**.

The behavior of **ALLOT** is more limited than allowed by the standard. **ALLOT** dynamically allocates the requested amount of memory and sets the parameter field address (PFA) of the *last created word* to the address of the alloted region. Thus, **ALLOT** *should always be preceded by* **CREATE**. Attempting to **ALLOT** without first creating a named dictionary entry, using **CREATE**, will result in a virtual machine error. Thus kForth limits the use of **ALLOT**, but code written for kForth will be portable to standard Forths.

The non-standard word **ALLOT?** is provided because kForth contains no **HERE** address. **ALLOT?** both reserves the requested memory *and* returns the starting address of the allotted memory region.

ALLOT? should be preceded by **CREATE** as described above. All memory is dynamically allocated, and freed upon exiting kForth.

3.8 String Operations

-TRAILING	`a u1 -- a u2`	reduce string length to ignore trailing spaces
/STRING	`a1 u1 n -- a2 u2`	`a2 = a1 + n, u2 = u1 - n`
BLANK	`a u --`	fill `u` bytes with the blank-space character starting at `a`
CMOVE	`a1 a2 u --`	move `u` bytes from source `a1` to dest `a2`
CMOVE>	`a1 a2 u --`	move `u` bytes from `a1` to `a2` in descending order
COMPARE	`a1 u1 a2 u2 -- n`	compare the strings `a1 u1` and `a2 u2`. Return zero if they are equal.
SEARCH	`a1 u1 a2 u2 -- a3 u3 b`	search for the string `a2 u2` within the string `a1 u1`; return **true** if found and the substring `a3 u3`
SLITERAL	`a u --`	compile a string address and count from the stack into the current definition

The following useful string words are provided in `strings.4th`.

SCAN	`a1 u1 n -- a2 u2`	search for first occurrence of character value `n` in the string specified by `a1 u1`. Return the substring `a2 u2` starting with the search character
SKIP	`a1 u1 n -- a2 u2`	search for first occurrence of character value not equal to `n`

See also Memory Operations.

3.9 Logic and Bit Manipulation Operations

AND	`n1 n2 -- n3`	bitwise AND of `n1` and `n2`
OR	`n1 n2 -- n3`	bitwise OR of `n1` and `n2`
XOR	`n2 n2 -- n3`	bitwise exclusive OR of `n1` and `n2`
NOT	`n1 -- n2`	one's complement of `n1`

INVERT	n1 -- n2	same as **NOT**
LSHIFT	n1 n2 -- n3	n3 is n1 shifted left by n2 bits
RSHIFT	n1 n2 -- n3	n3 is n1 shifted right by n2 bits
BOOLEAN?	x -- b	return TRUE if x is either TRUE or FALSE
.AND.	b1 b2 -- b3	logical AND of b1 and b2
.OR.	b1 b2 -- b3	logical OR of b1 and b2
.XOR.	b1 b2 -- b3	logical XOR of b1 and b2
.NOT.	b1 -- b2	b2 is the logical NOT of b1

The strict Boolean logic operators **.AND.** , **.OR.** , **.XOR.** , and **.NOT.** will throw an error if the input operand(s) are not Boolean. They can be useful in testing a program to ensure strict Boolean values are passed to logic predicates.

3.10 Arithmetic and Relational Operations

3.10.1 Single and Double Integer Operations

1+	n1 -- n2	increment: n2 = n1 + 1
1-	n1 -- n2	decrement: n2 = n1 - 1
2+	n1 -- n2	n2 = n1 + 2
2-	n1 -- n2	n2 = n1 - 2
2*	n1 -- n2	arithmetic left shift: n2 = n1*2
2/	n1 -- n2	arithmetic right shift: n2 = n1/2
CELLS	n1 -- n2	n2 is n1 times size in bytes of a cell
CELL+	n1 -- n2	n2 is n1 plus the size in bytes of a cell
FLOATS	n1 -- n2	n2 is n1 times size of a floating point number
FLOAT+	n1 -- n2	n2 is n1 plus the size of a floating point number
DFLOATS	n1 -- n2	n2 is n1 times size of double precision fp number
DFLOAT+	n1 -- n2	n2 is n1 plus size of double precision fp number
SFLOATS	n1 -- n2	n2 is n1 times size of single precision fp number
SFLOAT+	n1 -- n2	n2 is n1 plus size of single precision fp number
CHAR+	n1 -- n2	same as **1+**
+	n1 n2 -- n3	add
-	n1 n2 -- n3	subtract: n3 = n1 - n2
*****	n1 n2 -- n3	multiply
/	n1 n2 -- n3	divide: n3 = n1 / n2
+!	n a --	add n to value at address a
MOD	n1 n2 -- n3	modulus or remainder
/MOD	n1 n2 -- n3 n4	n3 = remainder and n4 = quotient for n1/n2
***/**	n1 n2 n3 -- n4	n4 = n1*n2/n3; intermediate value is 64 bit
***/MOD**	n1 n2 n3 -- n4 n5	n4 and n5 are remainder and quotient for n1*n2/n3
M+	d1 n -- d2	add single to double integer

M*	n1 n2 -- d	multiply two singles and return signed double
M*/	d1 n1 +n2 -- d2	multiply d1 by n1 to obtain triple cell result; then divide result by n2>0 to give signed double d2
UM*	u1 u2 -- ud	multiply unsigned singles and return unsigned double
UM/MOD	ud u1 -- u2 u3	divide unsigned double number by unsigned single and return remainder (u2) and quotient (u3). Returns −1 −1 for u2 and u3 on division overflow.
FM/MOD	d n1 -- n2 n3	divide double by single to give floored quotient n3 and *modulus* n2
SM/REM	d n1 -- n2 n3	divide double by single to give symmetric quotient n3 and *remainder* n2
DS*	d n -- t	multiply double and single to give signed triple length product
UDM*	ud u -- ut	multiply unsigned double and unsigned single to give unsigned triple length product
UTM/	ut u -- ud	divide unsigned triple by unsigned single to give unsigned double quotient
UTS/MOD	ut1 u1 -- ut2 u2	Divide unsigned triple ut1 by unsigned single u1 to give unsigned triple quotient ut2 and unsigned single remainder u2
STS/REM	t1 n1 -- t2 n2	Divide signed triple t1 by signed single n1 to give signed triple quotient t2 and signed remainder n2
D+	d1 d2 -- d3	double number addition
D−	d1 d2 -- d3	double number subtraction
ABS	n1 -- n2	absolute value
NEGATE	n1 -- n2	n2 = −n1
DABS	d1 -- d2	double number absolute value
DNEGATE	d1 -- d2	double number negation
MIN	n1 n2 -- n1 | n2	minimum of n1 and n2
MAX	n1 n2 -- n1 | n2	maximum of n1 and n2
DMIN	d1 d2 -- d1|d2	minimum of d1 and d2
DMAX	d1 d2 -- d1|d2	maximum of d1 and d2
=	n1 n2 -- b	test n1 equal to n2
<>	n1 n2 -- b	test n1 not equal to n2
<	n1 n2 -- b	test n1 less than n2
>	n1 n2 -- b	test n1 greater than n2
<=	n1 n2 -- b	test n1 less than or equal to n2
>=	n1 n2 -- b	test n1 greater than or equal to n2
U<	u1 u2 -- b	test unsigned u1 less than u2
U>	u1 u2 -- b	test unsigned u1 greater than u2
D=	d1 d2 -- b	test d1 equal to d2
D<	d1 d2 -- b	test d1 less than d2
DU<	ud1 ud2 -- b	test ud1 less than ud2
0<	n -- b	test n less than zero

0>	`n -- b`	test `n` greater than zero
0=	`n -- b`	test `n` equal to zero
0<>	`n -- b`	test `n` not equal to zero
D0=	`d -- b`	test `d` equal to zero
D0<	`d -- b`	test `d` less than zero
D2*	`d1 -- d2`	`d2` is the arithmetic left shift of `d1`
D2/	`d1 -- d2`	`d2` is the arithmetic right shift of `d1`
WITHIN	`n1|u1 n2|u2 n3|u3 -- b`	return **TRUE** if `n2|u2 <= n1|u1 < n3|u3`, given `n2|u2 < n3|u3`

kForth provides pre-defined constants **TRUE** (-1) and **FALSE** (0).

3.11 Floating Point Operations

3.11.1 Arithmetic and Relational Words

F+	`r1 r2 -- r3`	add
F-	`r1 r2 -- r3`	subtract: `r3 = r1 - r2`
F*	`r1 r2 -- r3`	multiply
F/	`r1 r2 -- r3`	divide: `r3 = r1/r2`
FABS	`r1 -- r2`	absolute value
FNEGATE	`r1 -- r2`	`r2 = -r1`
FROUND	`r1 -- r2`	round to nearest whole number
FTRUNC	`r1 -- r2`	truncate, towards zero, to whole number
FLOOR	`r1 -- r2`	truncate, towards minus infinity, to whole number
FMIN	`r1 r2 -- r1 | r2`	minimum of `r1` and `r2`
FMAX	`r1 r2 -- r1 | r2`	maximum of `r1` and `r2`
F0=	`r -- b`	test `r` equal to zero
F0<	`r -- b`	test `r` less than zero
F0>	`r -- b`	test `r` greater than zero
F=	`r1 r2 -- b`	test `r1` equal to `r2`
F<>	`r1 r2 -- b`	test `r1` not equal to `r2`
F<	`r1 r2 -- b`	test `r1` less than `r2`
F>	`r1 r2 -- b`	test `r1` greater than `r2`
F<=	`r1 r2 -- b`	test `r1` less than or equal to `r2`
F>=	`r1 r2 -- b`	test `r1` greater than or equal to `r2`

The following standard word is provided as Forth source in `ans-words.4th`:

F~	`r1 r2 r3 -- b`	test `r1` approximately equal to `r2`, within uncertainty `r3`; if `r3 = 0e`, `r1` and `r2` must be *exactly equal* in their binary representation

3.11.2 Floating Point Functions

F**	r1 r2 -- r3	r3 = r1 raised to power of r2
FSQRT	r1 -- r2	square root
FLOG	r1 -- r2	r2 = log base 10 of r1
FALOG	r1 -- r2	r2 = 10 raised to power of r1
FEXP	r1 -- r2	r2 = exp(r1)
FEXPM1	r1 -- r2	r2 = exp(r1) - 1
FLN	r1 -- r2	r2 = log base *e* of r1
FLNP1	r1 -- r2	r2 = log*e*(r1) + 1
DEG>RAD	r1 -- r2	degrees to radians
RAD>DEG	r1 -- r2	radians to degrees
FSIN	r1 -- r2	r2 = sin(r1)
FCOS	r1 -- r2	r2 = cos(r1)
FSINCOS†	r1 -- r2 r3	r2 = sin(r1); r3 = cos(r1)
FTAN	r1 -- r2	r2 = tan(r1)
FASIN	r1 -- r2	arc sine
FACOS	r1 -- r2	arc cosine
FATAN	r1 -- r2	arc tangent
FATAN2	r1 r2 -- r3	r3 is arc tangent of r1/r2 with proper quadrant
FSINH	r1 -- r2	r2 = sinh(r1)
FCOSH	r1 -- r2	r2 = cosh(r1)
FTANH	r1 -- r2	r2 = tanh(r1)
FASINH	r1 -- r2	inverse hyperbolic sine
FACOSH	r1 -- r2	inverse hyperbolic cosine
FATANH	r1 -- r2	inverse hyperbolic tangent

† In kForth-32, **FSINCOS** always uses the x87's native FSINCOS instruction. The returned sine and cosine values from **FSINCOS** may differ in accuracy from those returned by kForth's **FSIN** and **FCOS** words. **FSIN** and **FCOS** words will provide higher accuracy over a larger range of angles, while **FSINCOS** will be faster, in general.

3.12 Number Conversion

S>D	n -- d	convert single integer to double length integer
D>S	d -- n	convert signed double integer to signed integer
S>F	n -- r	convert single integer to floating point number
D>F	d -- r	convert double length integer to fp number
FROUND>S	r -- n	convert floating point to integer by *rounding*
FTRUNC>S	r -- n	convert floating point number r to integer n by *truncating* towards zero†

F>D	r -- d	convert fp number r to double integer d by truncating towards zero.
>FLOAT	a u -- r TRUE | FALSE	convert string to floating point number r. Return r and **TRUE** if successful, **FALSE** otherwise.
>NUMBER	ud1 a1 u1 -- ud2 a2 u2	convert digits of string a1 u1 and add this number to ud1 in the current base; result is ud2, and a2 u2 point to remaining part of string
NUMBER?	^str -- d b	convert counted string to signed double number in the current BASE; b is **TRUE** if successful
<#	ud -- ud	begin conversion of unsigned double to a string
#	ud1 -- ud2	convert the least significant digit of ud1 to a character; concatenate character to conversion string.
#S	ud1 -- 0 0	convert all significant digits in ud1 to string
SIGN	n --	attach minus sign to conversion string if n < 0
HOLD	n --	attach character with ASCII code n to the conversion string
#>	ud -- a u	drop the double number and return the string address and count

Other useful conversion words for number to string conversion and vice-versa, such as **F>FPSTR**, are given in Forth source in `strings.4th`.

†The word **FTRUNC>S** has the same behavior as the Forth-2012 word **F>S** ; however, **F>S** has previously been implemented in some Forth systems with *rounding* to single length signed integer behavior as well. Therefore, we have not implemented **F>S** in kForth, preferring instead that the rounding mode for conversions from floating point to single length signed integer be explicitly specified using either **FROUND>S** or **FTRUNC>S** . In contrast, the Forth-94 and Forth-2012 specifications call for **F>D** conversions to be truncating, and kForth's **F>D** conversion is compliant with those specifications. Explicit words to perform rounding to nearest and truncating conversions from floating point to double length signed integer may be defined as

```
: FROUND>D FROUND F>D ;
: FTRUNC>D F>D ;
```

3.13 Input and Output

BASE	-- a	return the address containing current number base
DECIMAL	--	set the number base to ten
BINARY	--	set the number base to two

HEX	--	set the number base to sixteen
KEY	-- n	wait for key press and return key code
ACCEPT	a n1 -- n2	read up to n1 characters into buffer a from keyboard. n2 is actual number input.
BL	-- 32	return the ascii value for a blank space character
WORD	n -- ^str	parse text from the input stream, delimited by character with ascii value n and return the address of a counted string containing the word
PARSE	n -- a u	parse text from the input stream, delimited by character n and return the parsed string
PARSE-NAME	-- a u	skip leading spaces and parse text delimited by a space. Return the parsed string.
CHAR	-- n	parse the next word, delimited by a space and return the ascii value of its first character
[CHAR]	-- n	version of **CHAR** for use in compile state
.	n --	display top item on the stack in the current base
.R	n u --	display n in the current base in u-wide field
U.	u --	display unsigned single u in current base
U.R	u1 u2 --	display u1 in the current base in u2-wide field
D.	d --	display signed double length number d
D.R	d u --	display signed double d in u-wide field
UD.	ud --	display unsigned double length number
UD.R	ud u --	display unsigned double ud in u-wide field
PRECISION	-- u	return the number of significant digits output by **FS.**
SET-PRECISION	u --	set the numer of significant digits output by **FS.**
FS.	r --	display the floating point number using scientific notation, with the number of significant digits specified by **PRECISION**
F.	r --	display the floating point number on top of the stack, using an automatic format.
.S	n1 n2 ... -- n1 n2 ...	non-destructive display of the data stack
."	--	parse text from the input stream, delimited by " and append the execution semantics to display the string within the current definition.
.(	--	parse and display text, delimited by ‘)’, from the input stream. The word is executed immediately.
CR	--	output carriage return
SPACES	n --	output n spaces
EMIT	n --	output character with ascii value n
TYPE	a u --	display u characters from buffer at a

SOURCE	-- a u	return address and count of the input buffer
REFILL	-- b	attempt to read another line from the input source and return flag
>FILE	--	change output stream from the console to a file. The filename is the next word in the input stream
CONSOLE	--	reset output stream to the console

The following non-standard output word is provided in `strings.4th`.

F.RD	r u1 u2 --	print a floating point number `r` in fixed point format with `u2` decimal places, right justified in a field of width `u1`.

The following standard terminal control words, and more, are provided in Forth source in `ansi.4th`:

PAGE	--	clear the screen and put cursor at top left.
AT-XY	n1 n2 --	position cursor at column `n1` and row `n2`. Origin is (0, 0).

3.14 File Access

OPEN	^name n1 -- n2	open file specified by counted string `^name` in mode `n1`, which can be the following: 0 read-only (**R/O**) 1 write-only (**W/O**) 2 read-write (**R/W**) `n2` is the file descriptor, a non-negative integer if successful.
LSEEK	n1 n2 n3 -- n4	change current position in opened file; `n1` is the file descriptor, `n2` is the offset, and `n3` is the mode with the following meaning: 0 offset is relative to start of file 1 offset is relative to current position 2 offset is relative to end of file `n4` is the resulting offset from the beginning of the file, or -1 if error.
READ	n1 a n2 -- n3	read `n2` bytes into buffer address a, from file with descriptor `n1`. `n3` is the number of bytes actually read.
WRITE	n1 a n2 -- n3	write `n2` bytes from buffer address a to file with descriptor `n1`. `n3` is the number of bytes actually written.
CLOSE	n1 -- n2	close file with descriptor `n1` and return status `n2` (0 if successful, -1 if error).
FSYNC	n1 -- n2	flush all buffered data written to file/device with descriptor `n1`. Return error code `n2`.
INCLUDE	--	parse the Forth source file name from the input stream and

	interpret the file.
INCLUDED a u -- ?	set the input stream for the interpreter to the specified file and process it line by line

The following standard file access words are provided as Forth definitions in `files.4th`:

R/O	-- n	"read-only" file access method
W/O	-- n	"write-only" file access method
R/W	-- n	"read-write" file access method
BIN	n1 -- n2	modify file access method for binary mode
CREATE-FILE	a u n1 -- n2 n3	create a file with name specified by string address and count a u, and access method n1. Return file descriptor n2 and result code n3
OPEN-FILE	a u n1 -- n2 n3	open an existing file, using access method n1, and return file descriptor n2 and result code n3
CLOSE-FILE	n1 -- n2	close the file with descriptor n1 and return result code n2
READ-FILE	a u1 n1 -- u2 n2	read u1 bytes into buffer at address a from file with descriptor n1 and return actual number of bytes read u2 and result code n2
WRITE-FILE	a u n1 -- n2	write u bytes from buffer a to file with descriptor n1; return result code n2
FILE-POSITION	n1 -- ud n2	return current file position ud and result code n2
REPOSITION-FILE	ud n1 -- n2	set file position to ud for file with descriptor n1 and return result code n2
FILE-SIZE	n1 -- ud n2	return the size of the file ud and the result code n2
FILE-EXISTS	^str -- b	return TRUE if the specified file exists
DELETE-FILE	a u -- n	delete the file specified by string a u, and return result code n
RENAME-FILE	a1 u1 a2 u2 -- n1	existing file name is specified by string a1 u1; new file name is specified by string, a2 u2.
READ-LINE	a u1 n1 -- u2 b n2	read a line of text, with at most u1 bytes, from file with descriptor n1 into the buffer a; return actual bytes read u2, success flag, and result code n2
WRITE-LINE	a u n1 -- n2	write a line of text having u bytes from buffer a into file with descriptor n1, and return result code n2
FLUSH-FILE	n1 -- n2	force buffered info for file with descriptor n1 to be written to disk. n2 is the result code.

3.15 Operating System Interface

SYSTEM	^str -- n	execute a shell command; ^str is the command line passed to the shell. Return code n is -1 on error, or the return value from the command.
SYSCALL	n1 ... nm m ncall -- nerr	perform system call ncall, with arguments n1 to nm, where 0<= m <=6
BYE	--	close the Forth environment and exit to the system.
CHDIR	^path -- n	change the current directory to the one specified in the counted string ^path. Return code n is OS dependent.
IOCTL	n1 n2 a -- n3	send device control request n2 to file with descriptor n1. Additional parameters are passed through buffer at address a. n3 is the status (0 if successful, -1 if error).
TIME&DAY	-- sec min hr day mo yr	return the local time
MS	u --	wait for u milliseconds
MS@	-- u	return number of milliseconds elapsed since start of kForth
US	u --	wait for u microseconds
US2@	-- ud	return number of microseconds elapsed since start of kForth
DLOPEN	azstr bflag -- nhandle	load the dynamic library file
DLERROR	-- azstr	return address of null terminated error string
DLSYM	nhandle azsym -- a	return address of symbol in library
DLCLOSE	nhandle -- nerr	close the dynamic library
FORTH-SIGNAL	xt n -- xtold	install Forth word as handler for signal n
RAISE	n -- ior	assert signal n
SET-ITIMER	n1 a1 a2 -- n2	set up timer signals
GET-ITIMER	n a -- n2	get timer countdown count
USLEEP	u --	wait for at least u microseconds

Numerous operating system functions are defined as Forth words in `syscalls.4th`. The use of **FORTH-SIGNAL** for handling signals is illustrated in the example Forth source files, `signals-ex.4th` and `sigfpe.4th`. The use of **IOCTL** for communicating with a device driver is illustrated in the Forth source file, `serial.4th`. The Forth source file, `dltest.4th`, provides an example of importing an external C library function, from a *shared object file,* into kForth using **DLOPEN**, etc. and calling the function from a Forth word.

3.16 Miscellaneous

kForth's **CALL** word provides a means for calling machine language procedures placed in *protected memory* (read-executable memory). The Forth source file, `mc.4th`, provides words for placing machine code into protected memory. See the file, `fcalls-x86.4th`, for an example of defining a Forth word to call machine code.

CALL	`a --`	call machine language subroutine at address `a`

4. Technical Information

1. Forth-94 Compliance
2. Threading Model
3. Signed Integer Division
4. Double Numbers
5. Floating Point Implementation
6. Special Features
7. Benchmarks and Tests
8. VM Error Codes
9. Source Code Map
10. Embedding kForth

4.1 Forth-94 Compliance

kForth-32 version 2.x is a subset of the Forth-94 standard (ANS Forth), given in DPANS94. Code may be written for kForth which is portable to standard Forth-94 systems with the use of trivially defined extensions (see the *Special Features* section below). The compliance with Forth-94 may be checked using John Hayes' suite of tests for the core words of an ANS Forth system: `tester.4th` and `core.4th`. Tests involving unsupported words such as **`HERE`** and **`,`** and **`C,`** have been commented out, as well as tests involving the **`BEGIN ... WHILE ... WHILE ... REPEAT ... THEN`** structure, and some weird variants of **`CREATE`** and **`DOES>`** usage. Compliance with the Forth-94 extension words for working with double length numbers may be checked using `dbltest.4th`. Tests are commented out for words which are not implemented in kForth.

Additionally, kForth-32 version 2.x incorporates concepts such as *name tokens*, and new words which have been specified to be part of the Forth-2012 standard. Examples of new Forth-2012 words included in kForth-32, 2.x, are **`SEARCH-WORDLIST`** , **`TRAVERSE-WORDLIST`** , **`NAME>STRING`** , **`NAME>COMPILE`** , **`NAME>INTERPRET`** , **`SYNONYM`** , and **`FTRUNC`**. kForth-32, version 2.x is intended to be transitional from the earlier 1.x versions to a future 3.x series which closely follows the Forth-2012 standard.

4.2 Threading Model

kForth is an *indirect threaded code* (ITC) system. The kForth compiler/interpreter parses the input stream into a vector of pseudo op-codes or Forth Byte Code. Upon execution, the vector of byte codes is passed on to a *virtual machine* which looks up the execution address of the words and performs either a *call* or an *indirect jump* to the next execution address. The type of threading used in the virtual machine is a **hybrid** of *indirect call threading* and *indirect jump threading*. The kForth virtual machine is implemented as a mixture of assembly language, C, and C++ functions. Only the assembly language portion of the virtual machine utilizes indirect jump threading.

4.3 Signed Integer Division

kForth implements *symmetric integer division*. An alternative form of signed integer division is called *floored integer division*. Both symmetric and floored division yield identical results when the two operands, dividend and divisor, are either both positive integers or both negative integers. However, when the two operands differ in sign, symmetric and floored integer division can give different results. For example,

Floored Division: `-8 3 / . -3 ok`

Symmetric Division: `-8 3 / . 2 ok`

Similarly, the word **MOD** yields different results on floored and symmetric division systems. Under floored division, **MOD** is truly a *modulus* operator (*i.e.* the result of *n1 n2* MOD is a number in the range [*0, n2*)), while under symmetric division, **MOD** simply returns a remainder. The following paper provides a discussion of integer division in computing languages: *Division and Modulus for Computer Scientists* by Daan Leijen.

Floored integer division was guaranteed by the Forth-83 standard. However, the Forth-94 standard revoked this guarantee and allowed system implementers to choose either symmetric or floored integer division. The rationale in revoking a fixed standard was to allow Forth systems to implement whatever form of integer division was best supported by the microprocessor hardware. Most microprocessors which provide signed integer division implement symmetric division. In kForth, the original rationale for using symmetric division was simply to maintain consistency with the GNU C implementation, which mandates the use of symmetric integer division per the ISO C99 standard (the symmetric version of **MOD** corresponds to the % operator in C). In general, floored division is considered by computer scientists and mathematicians to be the more useful form of signed integer division.

A significant problem with the Forth-94 standard is that, in practice, implementers of compliant Forth systems for a single hardware platform such as Intel x86 have chosen to use different forms of division. Consider the behavior of the Forth systems below, all running under Linux on a Intel PII:

```
gforth:   -8 3 MOD .  -2 ok
pfe:      -8 3 MOD .   1 ok
kforth:   -8 3 MOD .  -2 ok
iforth:   -8 3 MOD .  -2 ok
bigforth: -8 3 MOD .   1 ok
```

Therefore, a Forth program using signed integer division words (**/ MOD /MOD */MOD**) may produce different outputs under two different Forth-94-compliant systems. The Forth-94 standard addresses the portability issue by calling for use of the explicit floored and symmetric division words **FM/MOD** and **SM/REM** whenever it is important to explicitly specify the type of division. However, it is highly likely that Forth programmers will casually use signed integer division words such as **MOD** without always remembering the portability issue.

4.4 Double Numbers

kForth supports working with signed and unsigned double length numbers, and implements nearly all of the optional double number word set specified by Forth-94, either intrinsically or in the form of Forth source definitions (see `ans-words.4th` for the latter). In addition to the Forth-94 tests involving double numbers given in `core.4th`, further tests of double number words implemented in kForth are given in `system-test/dbltest.4th`.

4.4.1 Double Number Entry

One significant departure in kForth from typical Forth systems which provide double numbers is the method of entry of double length numbers. Traditional Forth recognizes the decimal point as a marker for a double number, e.g.

```
234.
```

is interpreted as a double number. *kForth does not permit double number entry in this manner*. The rationale behind this restriction is that such entries may easily be confused with floating point numbers. Such confusion will likely be common for new Forth users who have previously used other computer languages such as **C**. Even experienced Forth users who make frequent use of floating point calculations are also susceptible to such confusion. Since kForth-32, version 1.x and 2.x, uses the data stack to hold floating point numbers, and since a floating point number also occupies two stack cells (see next section), mistakes arising from misinterpreting entries with a decimal point may not be as readily apparent, leading to hard-to-find bugs.

4.4.2 kForth Method

The prohibition on standard double number entry in kForth demands that an alternate method be provided for entry of double numbers. This may be easily accomplished by using a string to double number conversion word. There are two ways to accomplish this. The first method is simple, but it is specific to kForth, while the second is more complex, but portable to other Forth-94/2012 systems. In the simple method, we may make use of the non-standard word, **NUMBER?**, to convert a counted string to a signed double length number, as follows.

```
c" -20123456789" NUMBER? DROP
```

NUMBER? actually returns a flag indicating whether or not the conversion succeeded, but we drop the flag in the above example for simplicity. If the conversion did not succeed, a double length zero will result.

4.4.3 Forth-94/2012 Compatible Method

The second method should be used if it is desired to port the code to other Forth systems. Forth-94 provides **>NUMBER** for converting a string to an unsigned double number. A more general string to double number conversion word, handling both signed and unsigned double numbers, may be written as follows.

```
variable dsign

: >d ( a u -- d|ud | convert string to a signed/unsigned double )
    0 0 2SWAP
    \ skip leading spaces and tabs
    BEGIN OVER C@ DUP BL = SWAP 9 = OR WHILE 1 /STRING REPEAT
    ?DUP IF
        FALSE dsign !
        OVER C@
        CASE
            [char] - OF TRUE dsign ! 1 /STRING ENDOF
            [char] + OF 1 /STRING ENDOF
        ENDCASE
        >NUMBER 2DROP
        dsign @ IF DNEGATE THEN
    ELSE DROP THEN ;
```

Using the above definition of `>D`, examples of double number entry are:

```
 s"  20123456789"  >d
 s" -20123456789"  >d
 s" +20123456789"  >d
```

Note that the method used above is not needed if the double number being entered fits within the bounds of a *signed* single number. Most cases of double number entry fit this scenario. In such a case, we may simply enter the single number, followed by **`S>D`**, e.g.

```
-234          S>D
 2147483647   S>D
-2147483649   S>D
```

4.5 Floating Point Implementation

The Forth-94 and earlier Forth standards allowed floating point numbers to be stored either on the *data stack* or on a separate *floating point stack*. kForth-32, 1.x and 2.x versions, uses the *data stack* for holding floating point numbers. The rationale for using the data stack for floating point operations in kForth was to allow legacy code written for earlier Forth systems (in particular the Forths from Laboratory Microsystems Inc.) to run without significant modifications under kForth. In kForth, a floating point number on the stack occupies two cells. Thus, under 32-bit Windows or Linux, floating point numbers are 64-bit double-precision numbers (equivalent to C's `double`).

The quality of the floating point arithmetic in kForth may be checked using the program, `paranoia.4th`, and other floating point tests provided in `forth-src/system-test.`

4.6 Special Features

Special features of kForth are described in a two-part article in Forthwrite magazine, issues **116** and **117**.These features are:

- The kForth dictionary is *dynamically allocated* as new definitions are added. Thus kForth does not implement a monolithic, fixed size dictionary, but can use as much memory as

provided by the host operating system. Several side effects result from using dynamic memory allocation to grow the dictionary:

- There is no **HERE** address in kForth.
- There is no **,** (comma operator) in kForth.
- There is no **C,** operator in kForth.

Owing to the fact that **HERE** does not exist, the word **ALLOT** not only allocates the requested amount of memory, but also has the non-standard behavior that it assigns the address of the new memory region to the *parameter field address* (PFA) of the last defined word. In kForth, the use of **ALLOT** must always be preceded by the use of **CREATE**. A variant of **ALLOT**, named **ALLOT?** is also provided. **ALLOT?** has the same behavior as **ALLOT** plus it returns the start address of the dynamically allocated region on the parameter stack. **ALLOT?** has the following equivalent definition under standard Forth:

```
: ALLOT? ( u -- a ) HERE SWAP ALLOT ;
```

ALLOT? is particularly useful in writing *defining words* in the absence of **HERE** and the comma operators. For example, to write your own integer constant defining word:

```
: CONST ( n -- ) CREATE 4 ALLOT? ! DOES> @ ;
```

or to write an address constant defining word (see below):

```
: PTR ( a -- ) CREATE 4 ALLOT? ! DOES> A@ ;
```

- kForth maintains *type stacks* corresponding to both the data and return stacks. The type stacks contain a type code for each corresponding data stack cell or return stack cell. This allows kForth to perform some rudimentary type checking, for example when an address is being accessed kForth verifies that the value's type is that of an address. Address values that are stored in variables must be retrieved with the word **A@** instead of **@** so that the type can be validated. Code written for kForth may be ported to other standard Forth implementations by defining **A@** as follows:

```
: A@ @ ;
```

- Unlike a conventional Forth interpreter which executes each token as it is interpreted, kForth continues to build up a vector of byte codes, until a keyword or end of line in the input stream necessitates execution. *Deferred execution* in interpreter mode is implemented by extending the normal concept of *precedence* in Forth. Instead of a single precedence-bit associated with each word, kForth uses a precedence-*byte* having *two* significant bits to

describe the behavior of each word in both compiled and interpreted modes. Thus, a word may have one of four possible precedence values:

0 not IMMEDIATE DEFERRED
1 IMMEDIATE DEFERRED
2 not IMMEDIATE NONDEFERRED
3 IMMEDIATE NONDEFERRED

To understand the execution behavior of a word in each of these states, it is helpful to view a table of execution modes for each precedence value and for the two compilation states: *interpret* and *compile*. We define the following execution modes:

- E0 – no execution, the opcode for the word is compiled into the opcode vector.
- E1 – execute current opcode vector up to and including current opcode.
- E2 – execute only current opcode and remove it from the opcode vector.

Precedence	Interpret	Compile
0	E0	E0
1	E2	E2
2	E1	E0
3	E1	E2

The ability to defer execution in interpreter mode allows “one-liners” to be executed from the kForth prompt without having to define a word. For example, the following line can be typed directly at the kForth prompt:

```
10 0 do i . loop
```

Ordinary Forth interpreters do not allow **do-loop**, **begin-while-repeat**, and **if-then** structures to occur outside of word definitions. kForth can interpret and execute such structures as long as they are completed on a single line of input.

Words which are nondeferred are those for which interpretation of the rest of the input line will depend on the execution of the word. Thus, the following intrinsic words in kForth have the nondeferred precedence attribute:

****	**.(**	**:**	**:NONAME**	**CREATE**
]	**'**	**WORD**	**PARSE**	**PARSE-NAME**
ALLOT	**CHAR**	**CONSTANT**	**2CONSTANT**	**FCONSTANT**
VARIABLE	**2VARIABLE**	**FVARIABLE**	**FORTH**	**ASSEMBLER**
WORDLIST	**DEFINITIONS**	**SET-ORDER**	**SET-CURRENT**	**ALSO**

ONLY	**PREVIOUS**	**[DEFINED]**	**[UNDEFINED]**	**FORGET**
DECIMAL	**HEX**	**SET-PRECISION**	**COMPILE,**	**INCLUDE**
INCLUDED	**SYNONYM**	**TO**	**VALUE**	**DEFER**
IS	**VOCABULARY**	**>FILE**	**CONSOLE**	**BINARY**
COLD	**COMPILE-NAME**	**#!**	**ALLOT?**	**ALIAS**

Only in very special cases will it be necessary for a programmer to use the **NONDEFERRED** keyword to set explicitly the interpretation precedence of a word. This is due to the automatic inheritance of the nondeferred attribute: if a word definition includes a nondeferred word, then the new word is automatically nondeferred also. Thus, for example, any word which has a definition including **WORD** is also a nondeferred word. Another example is a defining word, i.e. one which uses **CREATE**. Since **CREATE** is nondeferred the new defining word is also nondeferred.

The most common case in which the **NONDEFERRED** keyword should be explicitly used is in the definition of a word which changes the number base. For example,

```
DECIMAL
: BASE3 3 BASE ! ; NONDEFERRED
BASE3 21
```

If **BASE3** was not declared to be a nondeferred word, then `21` in the above line would be interpreted as decimal 21 rather than as decimal 7 (which is 21 in base 3).

- kForth can be started up in *debug mode* using the command line switch **-D**. Compiled op-codes and other debugging information are displayed in this mode. It is useful primarily for programmers interested in extending and debugging their own versions of kForth.

4.7 Benchmarks and Tests

Versions of standard benchmark programs for measuring kForth execution speed may be found in the subdirectory, `forth-src/benchmarks`. Forth source files in `forth-src/system-test` provide tests for compliance of core and standard extension words in Forth-94/2012, for words which are specific to kForth, and for floating point arithmetic. The tests require one of the following test harnesses: `ttester.4th` or `tester.4th`.

```
core.4th
coreplus.4th
memorytest.4th
filetest.4th
searchordertest.4th
stringtest.4th
```

```
dbltest.4th
to-float-test.4th
regress.4th
asm-x86-test.4th
divtest.4th
fatan2-test.4th
ieee-fprox-test.4th
ieee-arith-test.4th
fpzero-test.4th
fpio-test.4th
paranoia.4th
```

4.8 Exceptions and VM Error Codes

Non-zero return codes from the virtual machine (VM) follow the standardized *throw* codes specified in Forth-94 and Forth-2012 (see Table 9.1 in the specification). Reserved throw codes in Forth-2012 fall within the range -1 to -255. System-specific throw codes are allowed in the range, -256 to -4095. kForth-32 uses the system-specific throw codes shown in the table below.

Code	Exception
-256	Value on the stack did not have expected type `addr`
-257	Value on the stack did not have expected type `ival`
-258	Return stack was corrupted
-259	VM encountered invalid opcode
-260	**`ALLOT`** failed – cannot reallot memory for a word
-261	Failed on **`CREATE`**
-262	End of string not found
-263	No matching **`DO`**
-264	No matching **`BEGIN`**
-265	**`ELSE`** without matching **`IF`**
-266	**`THEN`** without matching **`IF`**
-267	**`ENDOF`** without matching **`OF`**
-268	**`ENDCASE`** without matching **`CASE`**
-269	Address outside of stack space
-270	Division overflow
-271	Unsigned double number overflow
-272	Incomplete **`IF … THEN`** structure

-273	Incomplete **`BEGIN`** structure
-274	Incomplete **`LOOP`** structure
-275	Incomplete **`CASE`** structure
-276	End of definition with no beginning
-277	Not allowed inside colon definition
-278	Unexpected end of input stream
-279	Unexpected end of string
-280	VM returned unknown error

4.9 Source Code Map

Source code for kForth-32 consists of the following C++, C, and assembly language files:

```
kforth.cpp
ForthCompiler.cpp
ForthVM.cpp
vmc.c
vm32-common.s
vm32.s
vm32-fast.s
fbc.h
ForthWords.h
ForthCompiler.h
ForthVM.h
kfmacros.h
VMerrors.h
```

A `Makefile` is used to build the executables. The kForth source code is made available to users under the GNU Affero General Public License (AGPL). The Linux version is provided as source code only and must be built locally on the user's machine (see installation). Under Linux, the standard GNU assembler, GNU C and C++ compilers, and the C++ Standard Template Library (STL) are required to build the executable.

4.10 Embedding kForth

The file `kforth.cpp` serves as a skeleton C++ program to illustrate how the kForth compiler and virtual machine may be embedded in a standalone program. XYPLOT is a more complex GUI program which embeds kForth to allow user extensibility. The file `xyplot.cpp` shows how to set

up hooks for calling C++ functions in the host program from the embedded kForth interpreter and vice-versa.