Command : loop
Type : compiler / flow control
Arguments : <entry>
Returns :
Description
encloses the address of the control directive in the dictionary and calculates the difference between <entry> and the current dictionary pointer. It then encloses this offset in the dictionary as the jump offset.

Command : +loop
Type : compiler / flow control
Arguments : <entry>
Returns :
Description
encloses the address of the +loop control directive in the dictionary, and then does the same as loop.

Command : leave
Type : compiler / flow control
Arguments :
Returns :
Description
encloses the address of the control directive in the dictionary.

Command : begin
Type : compiler / flow control
Arguments :
Returns : <entry>
Description
pushes the address of the next dictionary location to the stack.

Command : [
Type : compiler / string output
Arguments :
Returns :
Description
encloses the address of the control directive in the dictionary and then tokenises the characters up to the next ] into the dictionary space.
When the [ is executed the string between the left and right brackets [] will be echo directly to the screen.

Command : {
Type : compiler / string output
Arguments :
Returns : <string>
Description
encloses the address of the control directive in the dictionary and tokenises the buffer up to the next } into the dictionary space.
When then { is executed it will push the string between the left and right brackets {} onto the stack ( similar to the () in execute mode ).

Command : end,
Type : compiler / definitions
Arguments : <entry> <entry>
Returns :
Description
encloses the address of the program control directive in the dictionary, computes the relative jump value and writes it in the dictionary. Used in defining new compiler keywords.

Command : else
Type : compiler / flow control
Arguments : <entry>
Returns : <entry>
Description
else encloses the address of the control directive in the dictionary, calculates the value of the relative jump and saves the current value of the dictionary on the stack. Used as part of an if..else..then clause.

Command : then
Arguments : <entry>
Returns :
Description
pops the stack, computes the difference between the address and the current values of the dictionary pointer and store the value in the dictionary.

Command : do
Arguments : <start> <end>
Returns :
Description
encloses the address of the control direct in the dictionary and pops the top two entries from the data stack and pushes them onto the return stack.

Command : do,
Type : compiler / flow control
Arguments : <entry>
Returns : <entry>
Description
stores the top stack entry in the dictionary and pushes the new value of the dictionary pointer to the stack.

Command : if
Type : compiler / flow control
Arguments :
Returns : <entry>
Description
stores the address of the control directive in the dictionary, reserves the next word in the dictionary and pushes its address to the stack.

Command : while
Type : compiler / flow control
Arguments : <entry> <entry>
Returns :
Description
encloses the address of the control directive in the dictionary, the difference between the second stack entry and the dictionary pointer is calculated and enclosed as the jump offset, it also writes this offset into the address pointed to by the top stack entry.

Command : vdu
Type : command / IO / graphics
Arguments : <code>
Returns :
Description
The <code> will be sent to the vdu drivers - this command is similar to echo.

Command : plot
Type : command / IO / graphics
Arguments : <code> <x> <y>
Returns :
Description
This command will pass the stack entries to the vdu drivers, it assumes that the correct number of stack entries is present. If not then the stack will underflow and indeterminate data will be passed to the drivers.

Command : colour
Type : command / IO / graphics
Arguments : <colour>
Returns :
Description
passes the following control codes to the vdu drivers. 17 <colour> to select the text colour. vdu could be used but is less efficient.

Command : gcol
Type : command / IO / graphics
Arguments : <effect> <colour>
Returns :
Description
used to select the type of plotting performed by the graphics commands.

Command : ;
Type : compile mode terminator
Arguments :
Returns :
Description
semi-colon is used to terminate the definition of a new word, it exits from compile mode and enters execution mode. It writes the address of the interpreter routine semi into the dictionary and advance the pointer by 4.

Command : ;code
Type : compile mode terminator
Arguments :
Returns :
Description
semi-colon-code is used to terminate a forth word definition and encloses the address of the hidden routine scode in the dictionary. Machine code or assembler follows the statement and will be executed when the defined keyword is invoked.

Command : end
Type : compiler directive / flow control
Arguments : <entry>
Returns :
Description
End is used to terminate a begin end construct. The address on the stack was the value of the dictionary pointer when begin was executed.

Command : #ptr
Type : command / filing
Arguments : <handle>
Returns : <offset>
Description
#ptr will return the current position of the file pointer within the file selected by <handle> and push it onto the data stack.

Command : putc
Arguments : <data> <handle>
Returns :
Description
putc will write a lsb of <data> into the file selected by <handle>. Care should be taken as all files are opened read/write and may be modified.

Command : getc
Type : command / filing
Arguments : <handle>
Returns : <data>
Description
getc will read one byte from the selected file, expand it up to 32 bits and push it onto the stack.

Command : putw
Type : command / filing
Arguments : <data> <handle>
Returns :
Description
putw will write <data> as a 32 bit word into the file selected by <handle>, it is written lsb first.

Command : getw
Type : command / filing
Arguments : <handle>
Returns : <data>
Description
getw will read 4 bytes from the file selected by <handle> and push them onto the stack as a 32 bit word. The bytes are read lsb first.

Command : eof
Arguments : <handle>
Returns : <status>
Description
returns 1 if end of file <handle> has been reached, -1 if a read has occurred past the end of file and 0 otherwise.

Command : save
Type : command / HOST image dump
Arguments :
Returns :
Description
This command creates a file called image in the current directory that is a mirror image of the state of the forth machine when the command was typed. This does not include stacks or multitasking status - just the dictionary.

Command : oscli
Type : command / HOST
Arguments : <string>
Returns :
Description
<string> is in ‘(’ format and is passed unmodified to the operating system.

| | |
|---|---|
| Command | : ( |
| Type | : command / string operator |
| Arguments | : |
| Returns | : loads of stuff - see description |

Description

This command will copy the string that is enclosed by the brackets () to the data stack. it will then push the length of the string onto the data stack. The string will be terminated by a 0 byte starting at the address after the length word has been popped.

| | |
|---|---|
| Command | : type |
| Type | : command / string display |
| Arguments | : <length> <loads of words marking string> |
| Returns | : |

Description

this command will print the next <length> bytes from the top of the data stack. after completion the stack has the string removed as is aligned to a word boundary.

| | |
|---|---|
| Command | : close |
| Type | : command / Filing |
| Arguments | : <file handle> |
| Returns | : |

Description

this command will close the file that has a file handle <file handle>. a zero value for <file handle> will close ALL system files - this could be dangerous.

| | |
|---|---|
| Command | : load |
| Type | : command / filing |
| Arguments | : <length> <string> |
| Returns | : |

Description

this command will load the forth file identified by the string that is on the top of the stack ( see the '(' entry earlier ).

| | |
|---|---|
| Command | : open |
| Type | : command / filing |
| Arguments | : <string> |
| Returns | : <status> [<handle>] |

Description

open takes a filename string ( in type format ) from the stack and tries to open the file with read/write or create access. This means if the file exists it may be modified, and if not it will be created. If an error occurs then a zero will be pushed to the stack otherwise a 1 and the handle are pushed to the stack.

| | |
|---|---|
| Command | : ptr# |
| Type | : command / filing |
| Arguments | : <offset> <handle> |
| Returns | : |

Description

ptr# will move the filepointer of the file <handle> up to position <offset> in the file, extending the file if need be. If 0 is used as the offset the file may be read from the start.

Command : constant
Type : defining word
Arguments : <entry>
Returns :
Description
creates a word length constant dictionary entry whose value is the top stack entry and whose name follows the constant keyword. When the keyword name is executed the value of the constant will be pushed to the data stack.

Command : variable
Arguments : <entry>
Returns :
Description
creates a word length variable entry whose value is the top stack entry and whose name follows the constant keyword. When the keyword name is executed the address of the keyword is pushed to the stack.

Command : immediate
Type : vocabulary management
Arguments :
Returns :
Description
delinks the last keyword from the CURRENT vocabulary and links it to the COMPILER vocabulary. Adds keywords to the compiler vocabulary.

Command : <builds
Type : defining word
Arguments :
Returns :
Description
creates a CONSTANT keyword definition with an initial value of 0. The name is the token following <builds in the input buffer. MUST be terminated by a DOES> keyword.

Command : does>
Type : program control
Arguments : <return stack entry1> <return stack entry2>
Returns :
Description
replaces the first word in the code body of the latest entry in the vocabulary with the top return stack entry and replaces the code address with the second return stack entry.

Command : vocabulary
Type : defining word
Arguments :
Returns :
Description
creates a keyword whose name follows it , with an initial link to the latest entry in the CURRENT vocabulary. When executed it sets CURRENT to the link address. Used for defining new vocabularies.

Command : c?
Type : command / IO
Arguments :
Returns :
Description
this command will display the byte at the address pointed to by <entry> in the current number base on the display.

Command : forget
Type : dictionary management
Arguments :
Returns :
Description
Searches the current vocabulary for the keyword following the forget and will remove all keyword from that point forwards. It is to forget priority to completely erase the ARMFORTH dictionary.

Command : ‘
Type : command / vocabulary
Arguments :
Returns : [<entry>]
Description
scans the token following the tick in the input buffer and searches the CURRENT and CONTEXT vocabularies. If the word is found its execution address is pushed to the stack otherwise the keyword followed by a question mark will be displayed.

Command : definitions
Type : command / vocabulary
Arguments :
Returns :
Description
sets the system variable CURRENT to the value in CONTEXT.

Command : create
Type : defining word
Arguments :
Returns :
Description
creates a dictionary header for the primitive keyword whose name follows create and links it to the vocabulary.

Command : :
Type : compiling word
Arguments :
Returns :
Description
creates a token for the keyword following the : in the buffer, links it to the current vocabulary and sets the system in compile mode.

Command : next
Type : compiling word
Arguments :
Returns :
Description
encloses a jump to the inner interpreter NEXT routine in the dictionary.

Command : space
Type : command / IO
Arguments :
Returns :
Description
prints a space to the screen.

Command : echo
Type : command / IO
Arguments : <ascii code>
Returns :
Description
pops the stack and echoes the lsb of the word to the display.

Command : cls
Type : command / IO
Arguments :
Returns :
Description
clears the screen. It is a more efficient version of 12 echo.

Command : pos
Type : command / IO
Arguments :
Returns : <ypos> <xpos>
Description
pushes the current position (x,y) of the text cursor onto the stack.

Command : tab
Type : command / IO
Arguments : <ypos> <xpos>
Returns :
Description
moves the text cursor to position (xpos,ypos) on the screen.

Command : malloc
Type : command / memory
Arguments : <size>
Returns : <status> [ <address> ]
Description
malloc will try to allocate a space of <size> bytes, if unsuccessful <status> will be 0 else <status> will be 1 and <address> will be the address of the allocated block.

Command : free
Type : command / memory
Arguments : <address>
Returns :
Description
free will deallocate the block pointed to be <address>. No error will be reported if it is unsuccessful.

Command : ?
Type : command / IO
Arguments : <entry>
Returns :
Description
this command will display the word at the address pointed to by <entry> in the current number base on the display.

Command : lrot
Type : command / stack
Arguments : <entry1> <entry2> <entry3>
Returns : <entry3> <entry1> <entry2>
Description
rotates the top 3 stack entries to the left by one slot.
Example : 1 2 3 lrot . . . gives 1 3 2

Command : rs>ds
Type : command / inter-stack
Arguments :
Returns : <top entry from return stack>
Description
pops the top entry from the return stack and pushes it onto the data stack. CARE is recommended as popping a return address without careful consideration will result in the forth interpreter crashing. YOU HAVE BEEN WARNED

Command : ds>rs
Type : command / inter-stack
Arguments: <entry>
Returns :
Description
pops <entry> from the data stack and pushes it onto the return stack. CARE is recommended here as a ds>rs without an rs>ds will destroy the return stack and could kill forth. YOU HAVE BEEN WARNED

Command : .
Type : command / IO
Arguments : <entry1>
Returns :
Description
displays the top stack entry in the currently selected base.

Command : .r
Type : command / IO
Arguments <entry1> <entry2>
Returns :
Description
displays the <entry2> in a field width of <entry1>

Command : cr
Type : command / IO
Arguments :
Returns :
Description
echoes a CR/LF to the screen.

Command : inkey
Type : command / IO
Arguments :
Returns : <status> [<key number>]
Description
inkey waits .01 of a second for a key to be pressed, if no key is pressed it pushes a 0 otherwise it pushes the key and 1.

Command : get
Type : command / IO
Arguments :
Returns : <char>
Description
get waits indefinitely until a key is pressed and then it pushes the ascii value of the key onto the stack.

|  |  |
|---|---|
| Command | : ?sp |
| Type | : command / stack |
| Arguments | : |
| Returns | : <entry> |

Description

pushes the value of the data stack pointer prior to the push onto that data stack.

|  |  |
|---|---|
| Command | : i |
| Type | : compile / loop index |
| Arguments | : |
| Returns | : <entry> |

Description

pushes the value of the inner most loop index onto the data stack.

|  |  |
|---|---|
| Command | : j |
| Type | : compile / loop index |
| Arguments | : |
| Returns | : <entry> |

Description

pushes the value of the 2nd inner most loop index onto the data stack.

|  |  |
|---|---|
| Command | : k |
| Type | : compile / loop |
| Arguments | : |
| Returns | : <entry> |

Description

pushes the value of the 3rd inner most loop index onto the data stack.

|  |  |
|---|---|
| Command | : -sp |
| Type | : command / stack |
| Arguments | : <entry1> |
| Returns | : |

Description

this command will reduce the depth of the stack by <entry1> bytes. CARE is recommended.... always leave SP on a word boundary. YOU HAVE BEEN WARNED

|  |  |
|---|---|
| Command | : +sp |
| Type | : command / stack |
| Arguments | : <entry1> |
| Returns | : |

Description

this command will effectively push <entry1> bytes onto the stack although no actual valid data will be present. Used mainly for reserving stack space. CARE: always ensure that the stack remains on a word boundary when you have finished. YOU HAVE BEEN WARNED.

|  |  |
|---|---|
| Command | : rrot |
| Type | : command / stack |
| Arguments | : <entry1> <entry2> <entry3> |
| Returns | : <entry2> <entry3> <entry1> |

Description

rotates the top three stack entries once to the right.

Example : 1 2 3 rrot . . . gives 2 1 3

Command : align
Type : command / memory
Arguments : <entry1>
Returns : <entry>
Description
replaces entry1 with the nearest value rounded up to a word boundary
Example : dec 5 align . - would display 8

Command : over
Type : command / stack
Arguments : <entry1> <entry2>
Returns : <entry2> <entry1> <entry2>
Description
over pushes the second stack entry onto the data stack

Command : 2over
Type : command / stack
Arguments : <entry1> <entry2> <entry3>
Returns : <entry3> <entry1> <entry2> <entry3>
Description
2over pushes the third stack entry onto the data stack.

Command : swap
Type : command / stack
Arguments : <entry1> <entry2>
Returns : <entry2> <entry1>
Description
swaps the top 2 stack entries

Command : 2swap
Type : command / stack
Arguments : <entry1> <entry2> <entry3>
Returns : <entry3> <entry2> <entry1>
Description
swaps the first and third stack entries.

Command : dup
Type : command / stack
Arguments :
Returns : <entry1> <entry1>
Description
pushes the top stack entry onto the stack.

Command : drop
Type : command / stack
Arguments : <entry1> <entry2>
Returns : <entry2>
Description
pops the top stack entry.

Command: 2dup
Type : command / stack
Arguments : <entry1>
Returns : <entry1> <entry1> <entry1>
Description
duplicates the top stack entry twice.

Command : ?rs
Type : command / stack
Arguments :
Returns : <entry>
Description
pushes the value of the current return stack pointer,

Command : c!
Type : command / memory
Arguments : <entry1> <entry2>
Returns :
Description
puts the lsb of the word entry2 into the address entry1

Command : +!
Type : command / memory
Arguments : <entry1> <entry2>
Returns :
Description
adds entry2 to the value at address entry1

Command : c+!
Type : command / memory
Arguments : <entry1> <entry2>
Returns :
Description
adds the lsb of entry2 to the byte at address entry1

Command : 0set
Type : command / memory
Arguments : <entry1>
Returns :
Description
zeros the word at address entry1

Command : 1set
Type : command / memory
Arguments : <entry1>
Returns :
Description
sets the word at address entry1 to 1

Command : c0set
Type : command / memory
Arguments : <entry1>
Returns :
Description
sets the byte at address entry1 to zero

Command : c1set
Type : command / memory
Arguments : <entry1>
Returns :
Description
sets the byte at address entry1 to 1

Command : @
Type : command / memory
Arguments : <entry1>
Returns : <entry>
Description
pushes the word at address entry1 onto the data stack

Command : c@
Type : command / memory
Arguments : <entry1>
Returns : <entry>
Description
pushes the byte ( expanded to 32 bits ) at address entry1 onto the data stack

|  |  |
|---|---|
| Command | : entry |
| Type | : command |
| Arguments | : |
| Returns | : <entry> |

Description
pushes the address of the first header word in the latest entry in the CURRENT vocabulary.

|  |  |
|---|---|
| Command | : here |
| Type | : command |
| Arguments | : |
| Returns | : <entry> |

Description
pushes the address of the next free dictionary location onto the data stack.

|  |  |
|---|---|
| Command | : ca! |
| Type | : command |
| Arguments | : <entry> |
| Returns | : |

Description
Stores the top stack entry in the word address location of the latest entry in the CURRENT vocabulary.

|  |  |
|---|---|
| Command | : aspace |
| Type | : command |
| Arguments | : |
| Returns | : 32 |

Description
pushes the ascii value of a space ( 32 ) onto the data stack.

|  |  |
|---|---|
| Command | : , |
| Type | : command |
| Arguments | : <entry> |
| Returns | : |

Description
encloses the top data stack entry in the dictionary and advances the dictionary pointer by 1 word.

|  |  |
|---|---|
| Command | : c, |
| Type | : command |
| Arguments | : <entry> |
| Returns | : |

Description
encloses the lowest byte of the top data stack entry in the dictionary and moves the dictionary pointer forward by 1. CARE IS RECOMMENDED !!!

|  |  |
|---|---|
| Command | : buffer |
| Type | : system variable |
| Arguments | : |
| Returns | : <address of the input buffer> |

Description
used to get characters from the input buffer

|  |  |
|---|---|
| Command | : ! |
| Type | : command / memory |
| Arguments | : <entry1> <entry2> |
| Returns | : |

Description
puts the word entry2 in the address entry1

Command : 0<
Type : command / control
Arguments : <entry1>
Returns : <entry>
Description
if entry1 < 0 then push 1 else push 0

Command : 0>
Type : command / control
Arguments : <entry1>
Returns : <entry>
Description
if entry1 > 0 then push 1 else push 0

Command : execute
Type : command
Arguments : <entry>
Returns :
Description
pops the top stack entry and will execute the ARMFORTH code at that address.

Command : abort
Type : system control
Arguments :
Returns :
Description
restarts the interpreter after a fault

Command : bin
Type : command
Arguments :
Returns :
Description
selects binary display mode

Command : oct
Type : command
Arguments :
Returns :
Description
selects octal display mode

Command : dec
Type : command
Arguments :
Returns :
Description
selects decimal display mode

Command : hex
Type : command
Arguments :
Returns :
Description
selects hex display mode

Command : tst
Type : command / logic
Arguments : <entry1> <entry2>
Returns : <entry>
Description
if bit number <entry1> in entry2 is set then push 1 else push 0

Command : ror
Type : command / logic
Arguments : <entry1> <entry2>
Returns : <entry>
Description
rotate <entry2> by <entry1> bits right and push result onto stack.

Command : >
Type : command / control
Arguments : <entry1> <entry2>
Returns : <entry>
Description
if entry2 > entry1 then push 1 else push 0

Command: >=
Type : command / control
Arguments : <entry1> <entry2>
Returns : <entry>
Description
if entry2 >= entry1 then push 1 else push 0

Command : <
Type : command / control
Arguments : <entry1> <entry2>
Returns : <entry>
Description
if entry2 < entry1 then push 1 else push 0

Command : <=
Type : command / control
Arguments : <entry1> <entry2>
Returns : <entry>
Description
if entry2 <= entry1 then push 1 else push 0

Command : =
Type : command / control
Arguments : <entry1> <entry2>
Returns : <entry>
Description
if entry2 = entry1 then push 1 else push 0

Command : <>
Type : command / control
Arguments : <entry1> <entry2>
Returns : <entry>
Description
if entry2 <> entry1 then push 1 else push 0

Command : 0=
Type : command / control
Arguments : <entry1>
Returns : <entry>
Description
if entry1 = 0 then push 1 else push 0

Command : minus  
Type : command / arithmetic  
Arguments : <entry1>  
Returns : -<entry1>  
Description  
changes sign of top stack entry

Command : min  
Type : command / arithmetic  
Arguments : <entry1> <entry2>  
Returns : <entry>  
Description  
leaves the smaller of the 2 top entries on the stack

Command : max  
Type : command / arithmetic  
Arguments : <entry1> <entry2>  
Returns : <entry>  
Description  
leaves the larger of the 2 top entries on the stack

Command : <<  
Type : command / arithmetic  
Arguments : <entry1> <entry2>  
Returns : <entry>  
Description  
leaves <entry2> left shifted by <entry1> bits on the stack

Command : >>  
Type : command / arithmetic  
Arguments : <entry1> <entry2>  
Returns : <entry>  
Description  
leaves <entry2> right shifted by <entry1> bits on the stack

Command : not  
Type : command / logic  
Arguments : <entry>  
Returns : <entry>  
Description  
if top stack entry is not zero then push 1 else push 0

Command : and  
Type : command / logic  
Arguments : <entry1> <entry2>  
Returns : <entry>  
Description  
replaces the top 2 stack entries by the bitwise AND of them.

Command : eor  
Type : command / logic  
Arguments : <entry1> <entry2>  
Returns : <entry>  
Description  
replaces the top 2 stack entries by the bitwise EOR of them

Command : or  
Type : command / logic  
Arguments : <entry1> <entry2>  
Returns : <entry>  
Description  
replaces the top 2 stack entries by the bitwise OR of them

Command : bic  
Type : command / logic  
Arguments : <entry1> <entry2>  
Returns : <entry>  
Description  
clears bits in entry2 that are set in entry 1

Command : 2/
Type : command / arithmetic
Arguments : <entry1>
Returns : <entry1> DIV 2
Description
divides to stack entry by 2

Command : 4+
Type : command / arithmetic
Arguments : <entry1>
Returns: <entry1>+4
Description
adds 4 to top stack entry

Command : 4-
Type : command / arithmetic
Arguments : <entry1>
Returns : <entry1>-4
Description
subtracts 4 from the top stack entry

Command : 4*
Type : command / arithmetic
Arguments : <entry1>
Returns : <entry1>*4
Description
multiplies to stack entry by 4

Command : 4/
Type : command / arithmetic
Arguments : <entry1>
Returns : <entry1> DIV 4
Description
divides top stack entry by 4

Command : +
Type : command / arithmetic
Arguments : <entry1> <entry2>
Returns : <entry1>+<entry2>
Description
adds top 2 stack entries and pushes the result

Command : -
Type : command / arithmetic
Arguments : <entry1> <entry2>
Returns : <entry2>-<entry1>
Description
subtracts top 2 stack entries and pushes result

Command : *
Type : command / arithmetic
Arguments : <entry1> <entry2>
Returns : <entry2>*<entry1>
Description
multiply produces a 32 bit result

Command : /
Type : command / arithmetic
Arguments : <entry1> <entry2>
Returns : <entry2>DIV<entry1> <entry2>MOD<entry1>
Description
divides 2nd entry by first and leaves DIV and MOD on stack

Command : abs
Type : command / arithmetic
Arguments : <entry1>
Returns : <entry1>
Description
if negative makes entry positive

Command : search
Type : command
Arguments : <address>
Returns : <status> [<address>]
Description
used to locate a key word code address. The dictionary pointed to be <address> will be searched. if the token is found then status will be 0 and the address of the routine will be pushed to the stack, otherwise a 1 will be pushed.

Command : 0
Type : command / arithmetic
Arguments :
Returns : 0
Description
pushes a zero onto the data stack

Command : 1
Type : command / arithmetic
Arguments :
Returns : 1
Description
pushes a 1 onto data stack

Command : -1
Type : command / arithmetic
Arguments :
Returns : -1
Description
pushes -1 onto the data stack

Command : 1+
Type : command / arithmetic
Arguments : <entry1>
Returns : <entry1>+1
Description
adds 1 to top stack entry

Command : 1-
Type : command / arithmetic
Arguments : <entry1>
Returns : <entry1>-1
Description
subtracts 1 from the top stack entry

Command : 2+
Type : command / arithmetic
Arguments : <entry1>
Returns : <entry1>+2
Description
adds 2 to top stack entry

Command : 2-
Type : command / arithmetic
Arguments : <entry1>
Returns : <entry1>-2
Description
subtracts 2 from the top stack entry

Command : 2*
Type : command / arithmetic
Arguments : <entry1>
Returns : <entry1>*2
Description
multiplies top stack entry by 2

Command : core
Type : system variable
Arguments :
Returns :
Description
Invokes the CORE vocabulary.

Command : quit
Type : command
Arguments :
Returns :
Description
exits from the forth environment.

Command : singletask
Type : command
Arguments :
Returns :
Description
stop multitasking

Command : multitask
Type : command
Arguments :
Returns :
Description
start multitasking

Command : fork
Type : command
Arguments : address of routine to fork
Returns :
Description
takes address of routine from stack and starts the task when the next quantam elapses.
Example: ‘ test fork - will run test as a background task

Command : taskblock
Type : command
Arguments :
Returns :
Description

Command : quantam
Type : system variable
Arguments :
Returns : address of multitask data block
Description
used by multitasking code to get access to task queues and status. enables utilities to be written to examine task status etc. DO NOT MODIFY THE DATA BLOCK THIS POINTS TO.

Command : token
Type : command
Arguments : <separator>
Returns :
Description
tokenises the next ‘word’ separated by <separator> into the dictionary space
Example : aspace token - tokenises next word in input buffer

Command : compiler
Type : system variable
Arguments :
Returns : <entry>
Description
Pushes the address of the compiler variable which points to the last entry in the COMPILER vocabulary

Command : context
Type : system variable
Arguments :
Returns : <entry>
Description
Pushes the address of the system CONTEXT variable to the stack.

Command : current
Type : system variable
Arguments :
Returns : <entry>
Description
Pushes the address of the current vocabulary variable.

Command : dp
Type : system variable
Arguments :
Returns : address of dictionary pointer variable
Description
used to locate the current dp value

Command: himem
Type : system variable
Arguments :
Returns : address of himem variable
Description
the word at this address is the value of the highest address that the dictionary pointer could increase towards before corrupting the stack space.

Command : lbp
Type : system variable
Arguments :
Returns : address of line buffer pointer
Description
used by inline and token for character entry and tokenisation

Command : mode
Type : system variable
Arguments :
Returns : address of mode variable
Description
indicates if system is in compile ( 1 ) or execute mode ( 0 )

Command : state
Type : system variable
Arguments :
Returns : address of state variable
Description
used with mode to decide if a word is compiled or executed

**BEGIN..IF..WHILE**

Provides a form of WHILE loop in ARMFORTH, the code between BEGIN and IF will be repeated while the top stack entry upon execution of the IF is NON zero, if it is true then the code between the IF and WHILE will be executed and a branch made back to the code following BEGIN. If the stack contained a zero when the IF evaluated the code following the WHILE while be executed.

Example: print keys while <space> is not pressed.

```
: test begin get dup 32 <> if vdu while [ DONE ] ;
test [[a]]a[[b]]b[[<SPACE>]] DONE OK
```

**BEGIN..IF..ELSE..WHILE**

This loop construct is essentially the opposite of the loop above. It executes the code between the BEGIN and THE IF while the top stack entry is false and terminates when it is true.

Example:

```
: test begin get dup 32 = if [ Done ] else vdu while ;
test [[a]]a[[b]]b[[<SPACE>]] DONE OK
```

# FORTH command list

Some Syntax:

<...> means item must be present / or will be created
[   ] means optionally created.

About this list:

| | |
|---|---|
| Command | : name of command |
| Type | : variable / command / compiling command |
| Arguments | : <entry1> <entry2> <entry3> .. <nth entry> |
| Returns | : <entry1> <entry2> <entry3> .. <nth entry> |

| | |
|---|---|
| Command | : editv |
| Type | : system variable |
| Arguments | : |
| Returns | : address of edit vector |

Description
used if a replacement ‘inline’ input routine is written.

| | |
|---|---|
| Command | : base |
| Type | : system variable |
| Arguments | : |
| Returns | : address of system base |

Description
used to change the current number entry mode base
Example: 16 mode ! - sets hex numeric entry

# Flow Control

This section will detail all the flow control commands used in ARMFORTH. In the examples a brief description of what the example will do will be given and then the actual code. Type exactly what you see written, a <RETURN> means press the RETURN key, everything after the <RETURN> up to the OK shows what ARMFORTH will produce, unless surrounded by [[ ]] which means type the contents.

**BEGIN..END**

This is the fastest and most primitive of ARMFORTH loop constructs. The code between the BEGIN and the END will be executed if the top stack entry is zero when the END is reached, at which point program execution will continue with the code following the END.

Example: wait for the user to press space then print DONE.

```
:wait begin get aspace = end [ DONE ] ;
wait <RETURN> [[A B C <SPACE> ]] OK
```

**DO ... LOOP**

The most basic of ARMFORTH loop commands. It takes 2 stack entries and starting at the top stack entry count up to the second stack entry. The loop will always execute at least once.  The is no real loop variable but a loop variable may be accessed by the index i,j or k depending upon the depth of nesting ( i is first ). The addition is made BEFORE comparison of the terminating condition.

Example:  to count from 0 to 9 displaying each number

```
: test cr 10 0 do i . loop ;
test <RETURN>
0 1 2 3 4 5 6 7 8 9 OK
```

**DO ... +LOOP**

similar to DO..LOOP but the stack entry before the execution of +LOOP is the increment.

Example: to count from 0 to 8 in 2's

```
: test cr 10 0 do i . 2 +loop ;
test <RETURN>
0  2  4  6  8  OK
```

**LEAVE**

This command will cause the currently executing do..loop or do..+loop to terminate ( it set the current count to be the finishing value ). It will not prevent a loop from executing less than once.

**IF..THEN**

Provides a simple way to conditionally execute code, if the top stack entry is NON zero when the if evaluates then the code between the IF and THEN is executed, if the stack entry was zero then execution skips to the code following the then.

Example: print BEEP if a 1 is the top stack entry.

```
: bleep if [ BEEP ] then ;
0 bleep <RETURN> OK
1 bleep <RETURN> BEEP OK
```

**IF..ELSE..THEN**

Provides facilities for executing one section of code OR another. If the top stack entry is true the code between the IF and ELSE is executed otherwise the code between the ELSE and the THEN will be executed. Program execution will continue with the code following the THEN in both cases.

Example: if top stack entry is 1 print TRUE else print FALSE.

```
: test if [ TRUE ] else [ FALSE ] then ;
1 test <RETURN> TRUE OK
0 test <RETURN> FALSE OK
```

## Data Types

ARMFORTH has only has two real data types, bytes and words, any others are created as they are needed and may be accessed by combining the two supported types.

All stack entries are 32 bits wide and may be used in any fashion the user requires, the only condition is that EVERY stack entry be WORD aligned after any created structure has been pushed onto it.

There are built in string printing support and compile commands but string manipulation routines such as strcpy are not, they are provided in the stringlib file though, and any additional special structure handling routines would have to be written.

The format of an ARMFORTH string is a length as the top stack entry with the string contained on the stack in 4 byte chunks starting with the first letter.

## File Handling

ARMFORTH has RISCOS file handling and uses commands that are very similar to BASIC. These are LOAD, OPEN, CLOSE, PTR, EXT, EXT#, EOF and CLOSE ( These commands are covered in more detail later on in this manual ).

ARMFORTH filing commands ( open and load ) expect a string on the stack when they execute, the remainder expect a file handle.

ARMFORTH has a file stack that may be nested 7 deep, this means that when loading a file it could be split into functional libraries ( like the editor with stringlib ). If more than 7 files are opened at one time ARMFORTH will abort and it is a good idea to restart and reduce the file depth.

## Multitasking

It is import to appreciate that when multitasking the tasks have separate data and stack spaces they share the same dictionary space. This means that any task which modified the dictionary could crash the other tasks.

This is not an unacceptable risk because the advantages far surpass this problem, for example multiple copies of the same code could be running, with different data. Processes may communicate with each other via common data in the dictionary ( semiphores may be implemented ), obviously the system could be abused by forking a process that would compile a file into the background with disastrous results.

Finally all tasks have equal priority and are run in a round robin type task queue and are swapped every 10ms ( or as close to that interval as possible ) - It should be noted that assember code will NOT multitask in forth until it exits back into the ARMFORTH kernel ( this is how semaphores may be implemented ).

# ARMFORTH

Based upon the language FORTH

Version 1.10

David Redman
33 Manor Road,
Potters Bar,
Herts.
EN6 1DQ

## Introduction

ARMFORTH is a FORTH like language, making extensive use of RPN notation and stacks. many of the commands contained in the language are identical to their FORTH counterparts. However there are some differences.

What ARMFORTH offers :

- Multitasking : - 8 separate processes.
  Each task has its own data, and return stacks.
- Dynamic memory allocation words.
  ( malloc and free ).
- File handling
- An inbuilt editor written in ARMFORTH.
- A WIMP version ( not internally multitasking)
  for rapid program development and testing in
  the DESKTOP environment.
- 32 bit stack and data manipulation
- An assembler ( SOON )

ARMFORTH is small, only 10K ( unsqueezed ) for the NON WIMP version and 11K for the WIMP. It is also faster than BASIC, as an example:

| Language | empty loop to 1 million |
|---|---|
| ARMFORTH: | 6 |
| BASIC: | 14 ( integer X%) |
| | 38 ( using x ) |
| 'C' | 2 (using integer) |

However the size of the code is also important, for the 'C' a mere 48K, for BASIC about 40bytes ( 40K counting ROM ) for ARMFORTH, 24 bytes ( 10K including KERNEL ).