Preliminary 06/02/06

FastFORTH

With FastBASIC

Rev 0.1A

Preliminary 06/02/06

Overview

Preliminary 06/02/06

Table of Contents

1. Introduction to FastFORTH

1.1. Notation Conventions

Code examples and syntax diagrams are given in `this font`. All uppercase words and symbol words are statement words and must be typed as shown. Lowercase text, with or without angle braces, are parameters or return values and will depend on the statement word being used. The angle braces should not be included, if present.

1.2. What is FastFORTH?

FastFORTH is a forth-like language optimized for high speed operation and ease of use.

1.3. Where to go in here

<todo: finish>

2. FastBASIC Translator

2.1. Overview

2.1.1. What is FastBASIC

FastBASIC is a line translator that takes basic syntax and translates it into forth words.

2.1.2. Why use FastBASIC

FastBASIC is easy to learn and easy to master in a short time.

2.2. The Basic Language

2.2.1. The Basics

<todo: finish>

2.2.2. Variables

`<name> VAR BYTE`
Name is used to reference a byte (8 bits) of ram.

`<name> VAR WORD`
Name is used to reference a word (16 bits) of ram.

2.2.3. Constants

`<name> CON <expression>`
Name is assigned the constant result of expression. Constants are generally used to make code more readable.

2.2.4. Labels

`<name>:`
A label is created by appending a ':' to the end of a word. This label is used, without the ':', in GOTO and GOSUB commands.

2.2.5. Expressions

An expression is a collection of sources and operators that evaluates to a single 16 bit number. Operators are evaluated from left to right unless parenthesis are used to change the order.
Examples:

```
11 + 5 * 2 = 16 * 2 = 32
11 + (5 * 2) = 11 + 10  = 21
```

2.2.6. Commands

Commands take some number of parameters, separated by commas, and perform some action.

2.3. Binary Operators

Binary operators take two operands - one on the left and one on the right. The words 'left' and 'right' in the following descriptions do not have any special meaning. They are used simply to illustrate the syntax of the operator.

2.3.1. +

`<left> + <right>`
Gives the sum of the operands.

2.3.2. -

```
<left> - <right>
```
Subracts right from left.

2.3.3. *

```
<left> * <right>
```
Multiplies the two operands.

2.3.4. */

```
<left> */ <right>
```
Multiplies the two operands and divides the 32-bit temporary result by 8 bits (256).

2.3.5. **

```
<left> ** <right>
```
Multiplies the two operands and divides the 32-bit temporary result by 16 bits (65536).

2.3.6. /

```
<left> / <right>
```
Divides left by right.

2.3.7. //

```
<left> // <right>
```
Divides left by right and returns the remainder.

2.3.8. AND

```
<left> AND <right>
```
Bit-wise ANDs the two operands. A bit will be one if it is one in both left and right.

2.3.9. OR

```
<left> OR <right>
```
Bit-wise ORs the two operands. A bit will be one if it is one in either left or right.

2.3.10. XOR

```
<left> XOR <right>
```
Bit-wise XORs the two operands. A bit will be one if is different in left and right.

2.3.11. MIN

```
<left> MIN <right>
```
Gives the smaller of left or right.

2.3.12. MAX

```
<left> MAX <right>
```
Gives the larger of left or right.

2.4. Unary Operators

Unary operator take one operand to the right of the operator.

2.4.1. -

Gives the negative of the operand. Used when no operand is found to the left of '-'.

2.4.2. ABS

Gives the absolute value of the operand.

2.4.3. NOT

Gives the bit-wise inverse of the operand.

2.4.4. SIN

Gives the sine of the operand. The operand is in binary radians (BRADS). The result is from -127 to 127.

2.4.5. COS

Gives the cosine of the operand. See SIN, above, for details.

2.4.6. RAND

Gives a psuedo-random number between 0 and the operand, not including the operand.

2.5. Comparison Operators

All comparison operators are binary operators. No special treatment is given comparison operators. They are evaluated from left to right with any other operators. They may also be used in any expression, not just in a conditional statement. TRUE is defined as -1 and FALSE is defined as 0.

2.5.1. =

Equal. Gives TRUE when the left operand equals the right operand and FALSE otherwise.

2.5.2. <>

Not equal. Gives TRUE when the left operand does not equal the right operand and FALSE otherwise.

2.5.3. <

Less than. Gives TRUE when the left operand is less than the right operand and FALSE otherwise.

2.5.4. <=

Less than or equal. Gives TRUE when the left operand is less than or equal to the right operand and FALSE otherwise.

2.5.5. >

Greater than. Gives TRUE when the left operand is greater than the right operand and FALSE otherwise.

2.5.6. >=

Greater than or equal. Gives TRUE when the left operand is greater than or equal to the right operand and FALSE otherwise.

2.6. Sources

Sources are used as operands to the operators (above). All variables may be used as sources. All constants are considered sources.

2.6.1. INx

Equal to the value of the input pin x. <todo: explain x>

2.7. Destinations

Destinations must be the first word on a line followed by an equals sign (=). Followed by an expression that will be loaded into the destination. All variables can be used as destinations.

2.7.1. OUTx

Sets the output pin x.

2.7.2. RANDSEED

Sets the psuedo-random number generator seed value.

2.8. Commands

2.8.1. IF … THEN

Conditional execution of statements. The statements can be any number of basic lines. Execution will continue with the line following the ENDIF unless one of the statements executed is a GOTO.

```
IF <condition> THEN
      <statements>
ENDIF
```

Executes the statements if condition is non-zero. Otherwise continues with the line following the ENDIF.

```
IF <condition> THEN
      <first statements>
ELSE
      <second statements>
ENDIF
```

Executes the first set of statements if condition is non-zero. Otherwise executes the second set of statements.

2.8.2. FOR … NEXT

```
FOR <var> = <start> TO <end>
      <statements>
NEXT
```

Executes the statements until var is not in the range defined by start and end, inclusive. If start is less than end then var will count down by 1 at NEXT. Otherwise var will count up by 1. Both start and end may be any expression.

```
FOR <var> = <start> TO <end> STEP <value>
      <statements>
NEXT
```

Executes the statements until <var> is not in the range defined by <start> and <end>, inclusive. If <start> is less than <end> then <var> will count down by <value> at NEXT. Otherwise <var> will count up by <value>. Each of <start>, <end>, and <step> may be any expression.

2.8.3. DO … LOOP

```
DO
      <statements>
LOOP
```

Executes the statements forever.

```
DO WHILE <condition>
      <statements>
LOOP
```

Executes the statements while <condition> is true. The loop may not execute even once.

```
DO
      <statements>
LOOP UNTIL <cond>
```

Executes the statements until <condition> is true. The loop will always execute at least once.

2.8.4. GOTO

```
GOTO <destination>
```

Jumps to the desination label.

2.8.5. GOSUB

```
GOSUB <destination>
```

Jumps to the destination label until a RETURN is encountered. Then execution continues on the line following the GOSUB.

2.8.6. RETURN

```
RETURN
```

Returns from a GOSUB. A RETURN without a GOSUB will cause a reset of the target board.

2.8.7. PAUSE

```
PAUSE <expression>
```

Waits for the specified number of milliseconds.

2.8.8. HIGH

```
HIGH <expression>
```

Sets the specified pin to a high level and makes the pin an output.

2.8.9. LOW

```
LOW <expression>
```

Sets the specified pin to a low level and makes the pin an output.

2.8.10. INPUT

```
INPUT <expression>
```

Makes the specified pin an input.

2.8.11. OUTPUT

```
OUTPUT <expression>
```

Makes the specified pin an output.

3. FastFORTH Tutorial

3.1. Why use FastFORTH?

FastFORTH is easy to learn, but can be difficult to master. Once mastered, however, it is a very powerful language that grows anew with each project.

3.2. Starting from Basic

3.3. Postfix Notation and the Stack

3.4. Creating your own Words

4. FastFORTH Reference

4.1. Overview

4.1.1. Differences from ANS FORTH

Changes

The dictionary is stored separate from the parameter field. FastFORTH is direct threaded. Standard files are used as source input, instead of pages. Output is for target CPU, not the CPU that runs the compiler. Interactive mode sends commands to the target board via RS-232.

Additions

Code and colon words may be forward referenced. The compiler performs several optimizations (see Appendix ? for details). '\' in strings for non-printing characters. Preprocessor. Project commands. The init table. Hardware library packages. The following words have been added to the compiler or core: FOR NEXT UNNEXT OFFSET RAM SETRAM ALIAS DATABYTES DATAWORDS RECURSE XORC! XOR! 2** 2// NOT -; [; :" >4< CHOOSE 0EXIT COPY FLASHC@ FLASH@ FREAD FWRITE MSDELAY .. U/DIGIT SIN COS RAND RANDSEED SETSEED RAND@ CTYPE D.S CSCAN CSCAN" NOTHING

Omissions

No vocabularies. No dictionary manipulation. Immediate words and the compiler interactions associated with them. The following words are not implemented: BUILDS DOES> C, IMMEDIATE J VARIABLE

4.1.2. Words included

Words that are part of the compiler or part of the core are included here. Words that are part of a package are included in the next chapter on library packages.

4.1.3. Reading a word description

Word descriptions start with the word, which must be typed exactly as shown. Followed by the stack effects in parenthesis: <word> (<enter> -- <exit>). Where <enter> is the item(s) that are passed to the word and <exit> is the result(s) returned by the word.

4.2. Projects

4.2.1. Selecting a project

Type 'PROJECT', press 'F5', or select 'Select Project' from the menu to select a new project. A dialog box pops up that allows you to select the project file. Selecting a basic file (.BAS extension) will enable the FastBASIC translator.

4.2.2. What is a project?

A project is the top level source file used to specify all the other file(s) used by the project. A project may also have package options and/or additional code. A very simple project might contain all of the project specific code directly in the project file.

4.2.3. What goes into a project?

Generally, the project contains, in order: any package options, the board core file, user code. The user code can be in separate files or directly in the project file. A basic project will generally have the user code in the project file.

4.3. Modes

4.3.1. FastFORTH Console

The main operating mode for FastFORTH.

4.3.2. FastBASIC Translator

Enabled when the project extension begins with 'B'. Translates each basic line into a forth line that is then sent to the forth compiler.

4.3.3. Terminal

A simple dumb terminal using the current COM port.

4.3.4. Programming

The mode used to program a target CPU with the current image.

4.4. Defining Words

Defining words may only be used outside of code or colon definitions.

4.4.1. :

`: <name>`
Defines a new high level forth word. The words following, up until the ';', '-;', or '[;', are compiled.

4.4.2. :"

`:" <name>`
Defines a new high level forth word. The words following, up until the ';', '-;', or '[;', are compiled. Additionally, when <name> is used the compiler will compile an inline string following <name>. All text up to the next '"' will be consider this string.

4.4.3. ;

Terminates the colon definition with an EXIT.

4.4.4. -;

Terminates the colon definition with a token that skips over the header for the next colon definition. This cause the current word to execute the following word also.

4.4.5. [;

Terminates the colon word, but does not output any tokens. This should be used after an AGAIN or CHOOSE to reduce overall code size.

4.4.6. USER

`<bytes> USER <name>`
Allocates the number of bytes specified in user ram and creates a CONST, with the name given, that points to the first byte.

4.4.7. RAM

`<bytes> RAM <name>`
Allocates the number of bytes specified in ram at the current ram pointer and creates a CONST, with the name given, that points to the first byte.

4.4.8. SETRAM

`<value> SETRAM`
Sets the current ram pointer to the value given.

4.4.9. CONST

`<value> CONST <name>`
Creates a dictionary entry that will place <value> on the stack when <name> is used. No code is generated and <name> will be treated like an inline literal.

4.4.10. OFFSET

```
<prev> <size> OFFSET <name> ( prev -- new )
```
Creates a dictionary entry the will add <prev> to the top of stack when <name> is used. Then <size> is added to <prev> and left on the stack. No code is generated until <name> is used.

4.4.11. 2CONSTANT

```
<dvalue> 2CONSTANT <name>
```
Creates a word that will place <dvalue> on the stack when <name> is used.

4.4.12. HERE>

```
HERE> <name>
```
Creates a CONST word with the value of HERE.

4.4.13. TABLE

```
TABLE <name>
```
Defines a new word that takes the top of stack and returns the indexed value from the data words that follow.

4.4.14. CTABLE

```
CTABLE <name>
```
Defines a new word that takes the top of stack and returns the indexed value from the data bytes that follow.

4.4.15. ALIAS

```
ALIAS <old> <new>
```
Creates a copy of the dictionary entry <old> with the name <new>.

4.5. Math Operations

4.5.1. +

```
+ ( n1 n2 -- n3 )
```
Adds n1 and n2 together and leaves the result, n3, on the stack.

4.5.2. -

```
- ( n1 n2 -- n3 )
```
Subtracts n2 from n1 and leaves the result, n3, on the stack.

4.5.3. MINUS

```
MINUS ( n1 -- n2 )
```
Returns the two's complement negative of n1.

4.5.4. AND

```
AND ( n1 n2 -- n3 )
```
Bitwise ANDs n1 and n2 together and leaves the result, n3, on the stack.

4.5.5. OR

```
OR ( n1 n2 -- n3 )
```
Bitwise ORs n1 and n2 together and leaves the result, n3, on the stack.

4.5.6. XOR

```
XOR ( n1 n2 -- n3 )
```
Bitwise XORs n1 and n2 together and leaves the result, n3, on the stack.

4.5.7. ABS

```
ABS( n1 -- n2 )
```
Returns the absolute value of n1.

4.5.8. +-

```
+- ( n1 n2 -- n3 )
```
If n2 < 0 then n3 = -n1 else n3 = n1

4.5.9. 2**

```
2** ( n1 n2 -- n3 )
```
Multiply n1 by 2^n2 and leave the result, n3, on the stack.

4.5.10. 2//

```
2// ( n1 n2 -- n3 )
```
Divide n1 by 2^n2 and leave the result, n3, on the stack.

4.5.11. D+

```
D+ ( d1 d2 -- d3 )
```
Adds d1 and d2 together and leaves the result, d3, on the stack.

4.5.12. D-

```
D- ( d1 d2 -- d3 )
```
Subtracts d2 from d1 and leaves the result, d3, on the stack.

4.5.13. DMINUS

```
DMINUS ( d1 -- d2 )
```
Returns the two's complement negative of d1.

4.5.14. DABS

```
DABS( d1 -- d2 )
```
Returns the absolute value of d1.

4.5.15. D+-

```
D+- ( d1 n2 -- d3 )
```
If n2 < 0 then d3 = -d1 else d3 = d1

4.5.16. U*

```
U* ( u1 u2 -- ud3 )
```
Multiply u1 and u2 together and leave the double cell result, ud3, on the stack.

4.5.17. M*

```
M* ( n1 n2 -- d3 )
```
Multiply n1 and n2 together and leave the double cell result, d3, on the stack.

4.5.18. *

```
* ( n1 n2 -- n3 )
```
Multiply n1 and n2 together and leave the result, n3, on the stack.

4.5.19. U/

```
U/ ( ud1 u2 -- u3 u4 )
```
Divide ud1 by u2 leaving the result, u4, and the remainder, u3.

4.5.20. /MOD

```
/MOD ( n1 n2 -- n3 n4 )
```
Divide n1 by n2 leaving the result, n4, and the remainder, n3, on the stack.

4.5.21. M/

```
M/ ( d1 n2 -- n3 n4 )
```
Divide d1 by n2 leaving the result, n4, and the remainder, n3, on the stack.

4.5.22. /

```
/ ( n1 n2 -- n3 )
```
Divide n1 by n2 and leave the result, n3, on the stack.

4.5.23. MOD

```
/MOD ( n1 n2 -- n3 )
```
Divide n1 by n2 leaving the remainder, n3, on the stack.

4.5.24. */MOD

```
*/MOD ( n1 n2 n3 -- n4 n5 )
```
Multiply n1 and n2 then divide by n3 leaving the result, n5, and the remainder, n4, on the stack.

4.5.25. */

```
*/ ( n1 n2 n3 -- n4 )
```
Multiply n1 and n2 then divide by n3 leaving the result, n4, on the stack.

4.5.26. U*/MOD

```
U*/MOD ( u1 u2 u3 -- u4 u5 )
```
Multiply u1 and u2 then divide by u3 leaving the result, u5, and the remainder, u4, on the stack.

4.5.27. U*/

```
U*/ ( u1 u2 u3 -- u4 )
```
Multiply u1 and u2 then divide by u3 leaving the result, u4, on the stack.

4.5.28. U/DIGIT

```
U/DIGIT ( ud1 u2 -- u3 ud4 )
```
Divide ud1 by u2 leaving the result, ud4, and the remainder, u3. Used by '#' to get each digit.

4.5.29. B>W

```
B>W ( b1 -- n2 )
```
Sign extend byte b1 to cell n2.

4.6. Stack Operations

4.6.1. DUP

```
DUP ( n1 -- n1 n1 )
```
Duplicates the top item on the stack.

4.6.2. DROP

```
DROP ( n1 n2 -- n1 )
```
Removes the top item from the stack.

4.6.3. NIP

```
NIP ( n1 n2 -- n2 )
```

Removes the second item from the stack.

4.6.4. SWAP

```
SWAP ( n1 n2 -- n2 n1 )
```
Swaps the top two items on the stack.

4.6.5. OVER

```
OVER ( n1 n2 -- n2 n1 n2 )
```
Copies the second stack item to the top.

4.6.6. ROT

```
OVER ( n1 n2 n3 -- n2 n3 n1 )
```
Moves the third stack item to the top.

4.6.7. PICK

```
PICK ( … n1 … n -- … n1 … n1 )
```
Copies the nth stack item to the top. The zeroth item is the one under n.

4.6.8. ROLL

```
ROLL ( … n1 … n -- … … n1 )
```
Moves the nth stack item to the top. The zeroth item is the one under n.

4.6.9. ><

```
>< ( n1 -- n2 )
```
Swaps the bytes of the top of stack.

4.6.10. >4<

```
>4< ( n1 -- n2 )
```
Swaps the nibbles in the lower byte of the top of stack. The upper byte is unaffected.

4.6.11. 2DUP

```
2DUP ( d1 -- d1 d1 )
```
Duplicates the top double cell item on the stack.

4.6.12. 2DROP

```
2DROP ( d1 d2 -- d1 )
```
Removes the top double cell item from the stack.

4.6.13. 2SWAP

```
2SWAP ( d1 d2 -- d2 d1 )
```
Swaps the top two double cell items on the stack.

4.6.14. R

```
R ( R:n1 -- n1, R:n1 )
```
Copies the top of the return stack to the data stack.

4.6.15. >R

```
>R ( n1 -- R:n1 )
```
Moves the top of the data stack to the return stack.

4.6.16. R>

```
R> ( R:n1 -- n1 )
```
Moves the top of the return stack to the data stack.

4.6.17. 2>R

```
2>R ( d1 -- R:d1 )
```
Moves the top double cell item from the data stack to the return stack.

4.6.18. 2R>

```
2R> ( R:d1 -- d1 )
```
Moves the top double cell item from the return stack to the data stack.

4.6.19. RDROP

```
RDROP ( R:n1 n2 -- R:n1 )
```
Removes the top item from the return stack.

4.6.20. 2RDROP

```
2RDROP ( R:d1 d2 -- R:d1 )
```
Removes the top double cell item from the return stack.

4.7. Literals and Literal Operations

4.7.1. Inline literals

If a word cannot be found in the dictionary then the FORTH compiler will attempt to convert it to a number using the current base.

4.7.2. Inline string literals

<todo: finish>

4.7.3. TRUE

Puts -1 on the data stack.

4.7.4. FALSE

Puts 0 on the data stack.

4.7.5. 1+

Adds 1 to the top of stack.

4.7.6. 2+

Adds 2 to the top of stack.

4.7.7. 1-

Subtracts 1 from the top of stack.

4.7.8. 2-

Subtracts 2 from the top of stack.

4.7.9. 2*

Multiplies the top of stack by 2.

4.7.10. 3*

Multiplies the top of stack by 3.

4.7.11. 4*

Multiplies the top of stack by 4.

4.7.12. 6*

Multiplies the top of stack by 6.

4.7.13. 16*

Multiplies the top of stack by 16.

4.7.14. 2/

Divides the top of stack by 2.

4.7.15. 4/

Divides the top of stack by 4.

4.8. Control Flow Structures

4.8.1. IF … ENDIF

```
<flag> IF <words> ENDIF
```

Executes the words between 'IF' and 'ENDIF' if flag is non-zero.

4.8.2. IF … ELSE … ENDIF

```
<flag> IF <words1> ELSE <words2> ENDIF
```

Executes the words between 'IF' and 'ELSE' if flag is non-zero. Otherwise, executes the words between 'ELSE' and 'ENDIF'

4.8.3. BEGIN … UNTIL

```
BEGIN <words> <flag> UNTIL
```

Executes the words between 'BEGIN' and 'UNTIL' until flag is non-zero. Always executes the words at least once.

4.8.4. BEGIN … AGAIN

```
BEGIN <words> AGAIN
```

Executes the words between 'BEGIN' and 'AGAIN' forever.

4.8.5. BEGIN … WHILE … REPEAT

```
BEGIN <flag> WHILE <words> REPEAT
```

Executes the words between 'WHILE' and 'REPEAT' while flag is non-zero. May not execute words even once.

4.8.6. CHOOSE

```
<value> CHOOSE <dest0> <dest1> …
```

Chooses the destination to jump to based on value.

4.8.7. EXIT

```
EXIT
```

Exits the current word and returns to the calling word.

4.8.8. 0EXIT

```
<flag> 0EXIT
```

Exits the current word and returns to the calling word if flag is zero.

4.8.9. EXECUTE

```
<xt> EXECUTE
```

Calls the word pointed to by xt.

4.8.10. RECURSE

```
RECURSE
```
Calls the current word recursively.

4.8.11. FOR … NEXT

```
<count> FOR <words> NEXT
```
Executes the words between ‘FOR’ and ‘NEXT’ count times. Will not execute words if count is 0.

4.8.12. UNNEXT

```
UNNEXT
```
Removes the current counter from the return stack. Must only be used when ‘EXIT’ing a ‘FOR’ loop.

4.8.13. DO … LOOP

```
<end> <start> DO <words> LOOP
```
Executes the words between ‘DO’ and ‘LOOP’ while the current counter is less than end. Always executes words at least once. If start = end then words will be executed a very large number of times. ‘LOOP’ adds 1 to the current counter then checks to see if it’s greater than or equal to end.

4.8.14. DO … +LOOP

```
<end> <start> DO <words> <value> +LOOP
```
Executes the words between ‘DO’ and ‘+LOOP’ while the current counter is less than end. Always executes words at least once. If start = end then words will be executed a very large number of times. ‘+LOOP’ adds value to the current counter then checks to see if it’s greater than or equal to end.

4.8.15. LEAVE

```
LEAVE
```
Sets the current counter to the end value. This will cause the loop to end when ‘LOOP’ or ‘+LOOP’ is encountered.

4.8.16. UNLOOP

```
UNLOOP
```
Removes the current counter and end from the return stack. Must only be used when ‘EXIT’ing a ‘DO’ loop.

4.9. Memory Operations

4.9.1. @, C@, 2@

```
<addr> @
```
Fetches 1 (C@), 2 (@), or 4 (2@) bytes from memory to the stack.

4.9.2. !, C!, 2!

```
<value> <addr> !
```
Stores value in 1 (C!), 2 (!), or 4 (2!) bytes of memory.

4.9.3. +!, +C!

```
<value> <addr> +!
```
Adds value to 1 (+C!) or 2 (+!) bytes of memory.

4.9.4. XOR!, XORC!

```
<value> <addr> XOR!
```
XORs value with 1 (XORC!) or 2 (XOR!) bytes of memory.

4.9.5. FILL

```
<addr> <count> <value> FILL
```
Fills count bytes of ram starting at addr with value.

4.9.6. COPY

```
<src> <dest> <count> COPY
```
Copies count bytes of ram from src to dest. Src and dest should not overlap.

4.9.7. FLASH@, FLASHC@

```
<addr> FLASH@
```
Fetches 1 (FLASHC@) or 2 (FLASH@) bytes from flash memory to the stack.

4.9.8. FREAD

```
<src> <dest> <count> FREAD
```
Reads count words from src flash to dest ram.

4.9.9. FWRITE

```
<src> <dest> <count> FWRITE
```
Writes count words from src ram to dest flash.

4.9.10. FERASE

<todo: finish>

4.10. Comparison Operations

4.10.1. 0=

```
0= ( n1 -- flag )
```
Returns TRUE if the top of stack is 0.

4.10.2. 0<

```
0= ( n1 -- flag )
```
Returns TRUE if the top of stack is less than 0.

4.10.3. =

```
= ( n1 n2 -- flag )
```
Returns TRUE if n1 equals n2.

4.10.4. <

```
< ( n1 n2 -- flag )
```
Returns TRUE if n1 is less than n2.

4.10.5. <=

```
<= ( n1 n2 -- flag )
```
Returns TRUE if n1 is less than or equal to n2.

4.10.6. >

```
> ( n1 n2 -- flag )
```
Returns TRUE if n1 greater than n2.

4.10.7. >=

```
>= ( n1 n2 -- flag )
```
Returns TRUE if n1 greater than oer equal to n2.

4.10.8. MIN

```
MIN ( n1 n2 -- n3 )
```
Returns the smaller of n1 and n2.

4.10.9. MAX

```
MIN ( n1 n2 -- n3 )
```
Returns the larger of n1 and n2.

4.10.10. ?DUP

```
?DUP ( n1 -- 0 | n1 n1 )
```
Duplicates the top of stack if it's not zero.

4.10.11. D=

```
D= ( d1 d2 -- flag )
```
Returns TRUE if d1 equals d2.

4.10.12. D<

```
D< ( d1 d2 -- flag )
```
Returns TRUE if d1 less than d2.

4.10.13. D>

```
D> ( d1 d2 -- flag )
```
Returns TRUE if d1 greater than d2.

4.11. Number Formatting

4.11.1. <#

```
<# ( -- )
```
Setup for number output.

4.11.2. HOLD

```
HOLD ( char -- )
```
Save char to the beginning of the string.

4.11.3.

```
# ( ud1 -- ud2 )
```
Output one digit from the double cell item to the string.

4.11.4. #S

```
#S ( ud1 -- 0. )
```
Output digits from the double cell item to the string until ud1 is 0.

4.11.5. SIGN

```
SIGN ( n1 ud1 -- ud1 )
```
Output sign from saved itemon the stack to the string.

4.11.6. #>

```
#> ( ud1 -- addr count )
```
Closes number output by dropping the double cell item and returning the starting address and count.

4.11.7. D.S

```
D.S ( d1 n2 -- addr count n5 )
```
Output d1 to string leaving string address and count of chars. N5 is the number of additional chars to make the string n2 chars wide.

4.12. Serial I/O

4.12.1. EMIT

```
EMIT ( char -- )
```
Outputs char to the serial port.

4.12.2. KEY

```
KEY ( -- char )
```
Waits for the serial port to be ready and returns the char from the serial port.

4.12.3. ?KEY

```
?KEY ( -- flag )
```
Returns TRUE if the serial port has a charater ready.

4.12.4. SPACE

```
SPACE ( -- )
```
Outputs a space to the serial port.

4.12.5. SPACES

```
SPACES ( n -- )
```
Outputs n spaces to the serial port.

4.12.6. ."

```
." <string>"
```
Outputs the string to the serial port.

4.12.7. CTYPE

```
CTYPE ( caddr -- )
```
Outputs the counted string at caddr to the serial port.

4.12.8. TYPE

```
TYPE ( caddr n -- )
```
Outputs n bytes from caddr to the serial port.

4.12.9. CR

```
CR ( -- )
```
Outputs a carriage return and line feed to the serial port.

4.12.10. .

```
. ( n -- )
```
Outputs the signed number n to the serial port.

4.12.11. U.

```
U. ( u -- )
```
Outputs the unsigned number u to the serial port.

4.12.12. D.

```
D. ( d -- )
```
Outputs the signed double cell number d to the serial port.

4.12.13. .R

```
.R ( n1 n2 -- )
```

Output n1 to serial port with space padding to make n2 chars wide.

4.12.14. D.R

```
D.R ( d1 n2 -- )
```
Output double cell item d1 to serial port with space padding to make n2 chars wide.

4.12.15. ..

```
.. ( u1 -- )
```
Output u1 to serial port in hex.

4.12.16. ?DLINK

```
?DLINK ( -- )
```
Checks serial port for link command and execute DLINK if found. Should be executed in the main loop.

4.12.17. DLINK

```
DLINK ( -- )
```
Processes commands from FastFORTH compiler. May be called directly to wait for PC.

4.12.18. UNLINK

```
UNLINK ( -- )
```
Exits the interpreter loop in DLINK and returns to normal code execution.

4.13. Misc Words

4.13.1. “

```
“ <string>”
```
Compiles an inline counted string.

4.13.2. R”

```
R”
```
Returns an address pointing to the inline string following the calling word. Use in :” words.

4.13.3. COUNT

```
COUNT ( caddr1 -- caddr2 count )
```
Convert counted string to address and count.

4.13.4. CSCAN”

```
CSCAN“ <string>” ( char -- pos )
```
Finds char in the inline string. Returns pos or zero if not found.

4.13.5. CSCAN

```
CSCAN ( char caddr -- pos )
```
Finds char in the counted string at caddr. Returns pos or zero if not found.

4.13.6. RAND

```
RAND ( u1 -- u2 )
```
Returns a psuedo random number between 0 and u1.

4.13.7. RAND@

```
RAND@ ( -- d1 )
```
Returns a psuedo random double cell number.

4.13.8. RANDSEED

```
RANDSEED ( u1 -- )
```
Sets the psuedo random number seed value to u1.

4.13.9. SETSEED

```
SETSEED ( d1 -- )
```
Sets the psuedo random number seed value to d1.

4.13.10. MSDELAY

Delays for the given number of milliseconds.

4.13.11. MSWAIT

Wait until the given time arrives. 'USE_MSTIMER' must be defined for this word to be available.

4.13.12. NOTHING

High level nop. Used in CHOOSE lists to do nothing for that location.

4.13.13. SP@

Returns the current data stack pointer.

4.13.14. RP@

Returns the current return stack pointer.

4.13.15. SP!

Resets the data stack pointer.

4.13.16. RP!

Resets the return stack pointer.

4.13.17. +INT

Enables interupts.

4.13.18. -INT

Disables interupts.

4.14. Misc Compiler Words

4.14.1. ?FREE

```
?FREE
```
Displays the free space in the image and user ram.

4.14.2. ?REF

```
?REF
```
Checks to see if all forward references are resolved.

4.14.3. VLIST

```
VLIST
```
Lists all the entries in the dictionary.

4.14.4. SEE

```
SEE <name>
```

Displays info on the word specified.

4.14.5. .ERR”

```
.ERR” <text>”
```

Displays a user error with the given text.

4.15. Preprocessor

4.15.1. #IF … #ENDIF

```
<value> #IF <words1> #ENDIF
```

Compiles words1 if value is non-zero.

4.15.2. #IF … #ELSE … #ENDIF

```
<value> #IF <words1> #ELSE <words2> #ENDIF
```

Compiles words1 if value is non-zero. Otherwise, compiles words2.

4.15.3. #IFDEF

```
#IFDEF <name> <words1> {#ELSE <words2} #ENDIF
```

Compiles words1 if name is in the dictionary. May be used with an optional ‘#ELSE’ clause.

4.15.4. #IFNDEF

```
#IFNDEF <name> <words1> {#ELSE <words2} #ENDIF
```

Compiles words1 if name is not in the dictionary. May be used with an optional ‘#ELSE’ clause.

4.15.5. //

Line comment. All text to the end of the line is ignored. A space is required after // to be reconized correctly.

4.15.6. (…)

Stack comment. All text between (and) is ignored. A spaced is required after (to be reconized correctly.

4.16. Commands

These commands work in either FORTH or BASIC mode. Each command must be entered on a line by itself.

4.16.1. PROJECT

Opens the dialog box for selecting a project. See the projects section for more details.

4.16.2. BUILD

Builds the current project.

4.16.3. PROG

Programs the target with the currently built project.

4.16.4. LINK

Attempts to link to the target board for interactive mode. If the link was successful, the title bar will say ‘Interactive’ after the project name. The target board will send ‘READY.’ to indicate it has stopped executing code and is waiting for interactive commands.

4.16.5. TERM

Switches to terminal mode. See the mode section for more details.

4.16.6. PORT

Cycles to the next COM port (1-4). If the port is available 'Opened COMx' will be shown. Otherwise, 'Could not open COMx' will be displayed. The port number will be saved until changed by this command.

4.17. Control Words

4.17.1. LOAD"

Loads the file name as specified up to the trailing ".

4.17.2. DECIMAL

Sets the current number base to 10.

4.17.3. HEX

Sets the current number base to 16.

4.17.4. BASIC

Sets the input syntax to BASIC mode by enabling the FastBASIC translator. Must be on a line by itself.

4.17.5. FORTH

Sets the input syntax to FORTH mode by disabling the FastBASIC translator. Must be on a line by itself.

4.17.6. [

Sets the compiler state to execute.

4.17.7.]

Sets the compiler state to compile.

4.17.8. .ORG

`<value> .ORG`
Sets the compile address (HERE) to value.

4.17.9. CODERANGE

`<end> <start> CODERANGE`
Sets the range for code output. Sets the compile address (HERE) to start+2. Used by ?FREE.

4.17.10. USERRANGE

`<end> <start> USERRANGE`
Sets the range for user ram. Sets the user ram position to start. Used by ?FREE.

4.18. Assembly

4.18.1. CODE

`CODE <name>`
Defines a assembly word. Sets the compiler mode to assemble.

4.18.2. ENDCODE

`ENDCODE`
Terminates an assembly word. The data compiled will depend on the target CPU. Sets the compiler mode to execute.

4.18.3. LABEL

`LABEL <name>`

Defines a assembly word. Sets the compiler mode to assemble.

4.18.4. VECTOR

`VECTOR <name>`

Defines a vector as an assembly word. Sets the compiler mode to assemble.

4.18.5. :VECTOR

`:VECTOR <name>`

Defines a vector as a high level word. Sets the compiler mode to compile.

4.18.6. HWVECTOR

`HWVECTOR <name>`

Defines a hardware vector as an assembly word. Sets the compiler mode to assemble. The hardware vector must be previously defined as a constant. The current compilation address is stored in the address pointed to by that constant.

5. Library Packages

5.1. Servo Output

5.1.1. Overview

Provides slew rate controlled servo outputs with minimal software overhead. The number of servos supported will depend on the board and servo package used. See the comments in the individual servo packages for details on resolution and servo count. Only one servo package may be used in a project at a time.

5.1.2. SERVO and (SERVO)

```
SERVO ( POS SNUM -- )
```
Slews servo SNUM to POS using the current slew rate for servo SNUM. POS is pulse width in microseconds for SERVO and in clock units for (SERVO).

5.1.3. SDIRECT and (SDIRECT)

```
SDIRECT ( POS SNUM -- )
```
Sets the pulse width of servo SNUM to POS with no slewing. POS is pulse width in microseconds for SDIRECT and in clock units for (SDIRECT). This should be used to turn the servo on or off. A pulse width of 0 yields no servo pulse.

5.1.4. SETSLEW and (SETSLEW)

```
SETSLEW ( RATE SNUM -- )
```
Sets the slew rate for servo SNUM to RATE. RATE is pulse width in microseconds for SETSLEW and in clock units for (SETSLEW). A slew rate of 0 will disable slew rate control for that servo. (ie SERVO and SDIRECT operate the same.)

5.1.5. ?SERVO

```
?SERVO ( SNUM -- FLAG )
```
Checks servo SNUM and returns TRUE if the servo has stopped slewing.

5.1.6. ?ALLSERVOS

```
?ALLSERVOS ( -- FLAG )
```
Checks all servos and returns TRUE if all servos have stopped slewing.

5.1.7. SERVOINIT

```
SERVOINIT ( -- )
```
Sets up the timer(s) and resets all servo outputs to off with no slew rate control. Must be called before any other servo commands are used. Automatically called by startup code if the standard format is used.

5.2. Analog to Digital Input

5.2.1. Overview

Provide polled access to the analog inputs.

5.2.2. ADRESULT

```
ADRESULT ( ch -- value )
```
Returns the last result from the selected channel.

5.2.3. ?ADREADY

```
?ADSERVO ( -- FLAG )
```
Returns TRUE if the analog results are all valid.

5.2.4. ADSTART

```
ADSTART ( ch -- )
```

Starts an analog conversion for the specified channel. If ch is -1 then all channel will be converted.

5.2.5. ADINIT

```
ADINIT ( -- )
```

Turns on the analog converter. Called automatically by the standard board cores.

6. Appendicies

6.1. FORTH Glossary

6.2. Optimizations

6.3. MSP Assembler

6.4. Writing FORTH and CODE words for BASIC

Index

<todo: add index>