MForth
Mickey's Forth Interpreter

This is the cheat sheet for mforth. MForth is a command line interpreter that runs on micro-controllers. The purpose of this interpreter is to provide a serial port interface to a dumb terminal program on a PC. This serial port interface is designed to allow the hardware or software engineer to test hardware or run test programs under manual control of the user.

MForth is based on the original Forth by Charles Moore. The original book is still a good place to start : http://www.forth.com/starting-forth/. Mforth is also based on C.H. Tings' eforth : http://www.offete.com/. MForth tries to be fairly close to the ANSI Forth standard as well.

Starting Forth is an easy read and a good way to get started. MForth looks a lot like any other Forth interpreter.

If you don't have the time to read ahead here are the basics:

Forth is both an interpreter and a compiler. You can do things by typing them in or you can compile macros and then execute them by hand. You can store forth code in a text file and send it to the interpreter with TeraTerm, etc.

Forth uses spaces for delimiters between commands and values. Forth commands are usually all upper case and forth is case sensitive. There is no rigid syntax as long as commands and numbers are separated by spaces or newlines.

Forth can work in any number base, but the basics are **`HEX`** and **`DECIMAL`**.

Forth uses a stack, any number typed while interpreting gets pushed on the stack. Thus
`1 2 3` <CR>
Pushes 1 2 and 3 on the stack. 3 is on the top of the stack (TOS).
The command **`.`** Removes and prints the top number on the stack. The numbers are interpreted in the number base that is currently active, and you can mix and match.
Thus **`HEX FF DECIMAL 255`** <cr> pushes 0xff and then 255 (also 0xFF) onto the stack.

The math and logic operator operate on the items on the stack. Thus this should be fairly obvious :
`1 1 + .`
Does indeed print “2” as a result.

Forth commands can take parameters, they are simply pushed on the stack before hand. There are two very handy forth words for reading and writing memory they are **`@`** and **`!`** (sometimes referred to as at and bang, or get and store). **`@`** expects the top of the stack to be a memory location that you want to read. **`@`** reads that location on the TOS. Thus the sequence **`HEX 1234 @ .`** <cr> puts forth in hex, pushes 1234 in hex, then reads memory location 1234 and puts the value on the stack. **`@`** and **`!`** read and write 32 bit values, there are other similar words **`W@`** (word at) and **`W!`** (word bang) read and write 16 values (the addresses are still 32 bits) and **`C@`** (char at) and **`C!`** (char bang) read and write 8 bit values.

`!` expects the TOS to contain a memory address, and the second item down the stack to be a value. The value is stored at the address. Thus `HEX 55 1234 !` <cr> stores the value 55 at hex 1234.

You can make macros of basically anything, for example to set a bit in an register you can write this :

```
HEX
: SET_BIT 8000 @ 4 OR 8000 ! ;
: CLR_BIT 8000 @ 4 NOT AND 8000 ! ;
```

This creates two forth commands (called words), one that sets bit 4 at location 8000 and one that clears the same bit. It's up to you to figure out how the values are on the stack are being manipulated to get the result I want. The best thing to do is sit down with the interpreter and try some of the commands. Note that the forth word `.S` prints out the contents of the stack without destroying it. The only commands that you **can't** use **outside** of a macro are basically `IF,ELSE,THEN`, `BEGIN,WHILE, REPEAT` and `DO,LOOP`, Note that the way you define a macro is start with `:`, then a space and macro name, then the stuff you want to do, then a space then a `;`. Note that when you put forth in HEX, the numbers in the macros you are compiling are in that number base. You can change number base between macro definitions (there is also a way to change base within a number definition, but that's a tiny bit harder).

You can execute these words from the command line as follows: `SET_BIT`<CR> or you can do both on the same line `SET_BIT CLR_BIT` <CR>.

Now lets suppose that you want to write a scope trigger loop with the two commands and just toggle the bit over and over. This is a standard forth program that will do that.

```
HEX
: SET_BIT 8000 @ 4 OR 8000 ! ;
: CLR_BIT 8000 @ 4 NOT AND 8000 ! ;
: TOGGLE
     BEGIN
       ?KEY 0=
       WHILE
          SET_BIT
          CLR_BIT
       REPEAT
;
```

When you run the `TOGGLE` word it will loop continuously calling `SET_BIT` then `CLR_BIT`. If you forget the name of something the word `WORDS` shows you all the words in the dictionary, the dictionary being the collection of all the forth commands. You can put comments on a line of forth by putting a (at the end of a line and closing with a). Note that the ('s and)' must have spaces around them. So the code above, commented for you understanding is as follows:

```
: TOGGLE ( define a new word called TOGGLE )
     BEGIN ( start a BEGIN, WHILE, REPEAT loop )
       ?KEY 0= ( ask if the user has hit a key, then invert it
       WHILE ( while expects a condition in this case true if no
             ( key has been pressed )
          SET_BIT   ( all this stuff up to repeat is done if while )
                    ( is true)
          CLR_BIT
       REPEAT ( this says go back to begin and try again )
; ( close the macro )
```

There are other forth control structures, this one shows how to use **IF**. Notice that the condition for the **IF** is passed to the word when you run it.

```
: DO_BIT
     IF
          SET_BIT
     ELSE
          CLR_BIT
     THEN
;
```

To run this you would type 1 DO_BIT to set the bit or 0 DO_BIT to clear the bit.
Finally a **DO LOOP.** Note again that the number of times to do the loop is passed in. **DO** expects two values on the stack, the starting and stopping index. We pass the stopping index in when the word is run, the lower index 0, is put on the stack when the word executes.

```
: DO_SOME_BITS
0 DO
     SET_BIT
     CLR_BIT
  LOOP
;
```

If you want to set and clear the bit 10 times type **10 DO_SOME_BITS**.

Most forth words are written with some documentation that explains what the word expects on the stack when you call it and what the word leaves on the stack when it exits. This is known as a stack diagram and is basically a comment with the list of parameters on the stack followed by – followed by the items left on the stack. Suppose we write a word to tell if a value is greater than 100, here it is with a stack diagram:

```
( Word GT10 determines if a number is greater than 10 )
( value – 1|0 )  ( <----- this is the stack diagram)
: GT10
   10 > IF 1 ELSE 0 THEN ;
```

Notice that this word shows how you can put something on the stack during a word.

You can output text messages in a word as well, by example:

```
: PRINT_HELLO ." HELLO WORLD" ;
```

Prints HELLO WORLD, when executed. Note that **."** is a command and needs a space before and after it but the closing " is simply a delimiter and doesn't need a space before it. If you need to output a carriage return the word CR will do that. Thus

```
: PRINT_HELLO CR ." HELLO WORLD" CR ;
```

Prints hello world on a blank line by itself.

Glossary

1.1 Variables, Constants and Defines

1.1.1 CONSTANT (value --)

Creates a word that when executed leaves a constant value on the stack.

Example:

```
1234 CONSTANT C1 // creates Constant C1
C1 . 1234 <OK>
```

1.1.2 VARIABLE (--)

Creates a single cell variable, set to 0 initially. When executed the name of the variable returns the address of the variable.

Example

```
VARIABLE myVar
1234 myVar !  // store 1234 in myVar
myVar @ .  1234 // prints the value in the variable.
```

1.1.3 ALLOT (#cells --)

Add space onto the most recently declared word. Used to add memory to a variable. Must be used immediately after a variable declaration and before any other words, constants or variables are declared. Does not set the cells to 0.

Example:

```
VARIABLE myArray  10 ALLOT // Creates a variable with 11 cells
```

1.2 Stack Manipulation

1.2.1 PICK (… item# -- … value at stack depth #)

Expects a number, n, on the top of the stack, makes a copy of the nth element in the stack, not counting the value n that was pushed on the stack. For example 0 PICK duplicates the top stack item. 1 PICK makes a copy of the 2nd item down in the stack, etc.

Example:

```
1 2 3 // stack is 1 2 3
2 PICK // stack is now 1 2 3 1
```

1.2.2 2DROP (… a b -- …)

Drops the top two item of f the stack.

1.2.3 NIP (a b c – a c)

Removes the second element down in the stack.

1.2.4 2DUP (a b – a b a b)

Duplicates the top two items on the stack.

1.2.5 OVER (a b c – a b c b)

Duplicates the second item down in the stack, leaving it on top of the stack.

1.2.6 DUP (a – a a)

Duplicate the top stack item.

1.2.7 SWAP (a b – b a)

Swap the top two stack items.

1.2.8 ROT (a b c – b c a)

Move the third item on the stack to the top. The first and second items become the second and third items.

1.2.9 RP0 (-- R0)
Place the starting address of the returns stack on the data stack.

1.2.10 SP0 (-- SP0)
Place the starting value of the data stack, on the stack.

1.2.11 RP! (V --)
Put the top item on the data stack into the Return Stack pointer.

1.2.12 R@ (-- v)
Use the current returns stack pointer to retrieve an item from memory (essentially load on the data stack the first item on the return stack).

1.2.13 RP@ (-- RP)
Place the current Return Stack pointer value on the data stack.

1.2.14 SP! (v --)
Write over the current data stack pointer with the value on the top of the stack.

1.2.15 DROP (a b c – a b)
Drop the top stack item.

1.2.16 SP@ (-- SP)
Leave the current value of the stack pointer on the data stack.

1.2.17 R> (-- v)
Pop the return stack onto the data stack.

1.2.18 >R (v --)
Pop the data stack onto the return stack

1.2.19 ?DUP
Duplicate the top stack item if it is non-zero.

1.3 Math/Logic Words

1.3.1 / (a b – a/b)
Divide the second stack item by the item on the top of the stack.

1.3.2 * (a b – a*b)
Multiply the top two stack items.

1.3.3 - (a b – a-b)
Subtract the top stack item from the next item up on the stack.

1.3.4 + (a b – a+b)
Add the top two stack items.

1.3.5 >> (a b – a>>b)
Shift the 1 deep stack item to the right by the number of bits specified by the value on the TOS.

1.3.6 << (a b – a<<B)
Shift the 1 deep stack item to the left by the number of bits specified by the value on the TOS.

1.3.7 MOD (a b – a%b)
Leave the remainder after division (modulus).

1.3.8 /MOD (a b – a/b a%b)

Leave the quotient and remainder on the stack as the result of an integer division.

1.3.9 XOR (a b – a^b)
Exclusive or the top two stack items.

1.3.10 NOT (a – ones complement of a)
Take the ones complement of the item on the top of the stack.

1.3.11 OR (a b – a|b)
Bitwise or the top two stack items.

1.3.12 AND (a b – a&b)
Bitwise and the top two stack items.

1.3.13 +! (v addr --)
Add the value 1 deep in the stack to the value stored at the address on top of the stack. Can be used to increment/decrement variables easily.

1.3.14 ABS (a – abs(a))
Take the absolute value of the top of the stack.

1.3.15 > (a b – T|F)
Compare the top two items, return true if the 2nd item is greater than the 1st item, else return false.

1.3.16 < (a b – T|F)
Compare the top two items, return true if the 2nd item is less than the 1st item, else false.

1.3.17 NEGATE (a -- -a)
Take the two’s complement of the top of the stack.

1.3.18 0= (v – T|F)
Compare the top of the stack to 0 and if so return true, else return false.

1.3.19 0> (v – T|F)
Return true if the item on the top of the stack is greater than 0..

1.3.20 0< (v – T|F)
Return true if the item on the top of the stack is less than 0.

1.3.21 = (a b – T|F)
Compare the top two stack items and return true if they are equal, else return false.

1.4 Memory Words

1.4.1 W! (v addr --)
Store the 16 bit value at the given address.

1.4.2 W@ (addr – v)
Read the 16 bit value at the address.

1.4.3 C@ (addr – v)
Read an 8 bit value at the address.

1.4.4 C! (v addr --)
Store the 8 bit value at the given address.

1.4.5 ! (v addr --)
Store the 32 bit value at the address.

1.4.6 @ (addr – v)
Read the 32 bit value at the given address.

1.4.7 CI>BA (addr i – addr')
Given an address and index, compute the byte address of the ith cell in the array represented by addr. Used to efficiently index arrays.

1.4.8 DUMP (addr count --)
Dump out the contents of memory at addr for count bytes.

1.5 I/O Words

1.5.1 ?KEY (-- T|F)
Return true if an input character is ready to be read, else return false. Predicts whether or not KEY will immediately return.

1.5.2 $" (--)
Inserts a string that is terminated with " into the current word definition. When executed, leaves the address of the string on the TOS. Used to form strings to send out to UARTS, etc..

1.5.3 ." (--)
Inserts a string that is terminated with " into the current word definition. When executed, prints the string on the console.

1.5.4 .S (--)
Non-destructively prints the current data stack contents.

1.5.5 . (a --)
Prints the value on the top of the stack using the current number base.

1.5.6 DECIMAL (--)
Changes to base 10 for the current input/output number base on the console.

1.5.7 HEX (--)
Changes to base 16 for the current input/output number base on the console.

1.5.8 #> (v – addr count)
Completes the formatting of a number for output. Returns the length of the string and the address of the string.

1.5.9 TYPE (addr count --)
Print a counted string on the console.

1.5.10 .ID (cfa --)
Given a pointer to base word object, print the name of the word.

1.5.11 COUNT (addr – addr+1 count)
Take a pointer to a counted string in memory and leave the count and the address of the string itself on the stack, suitable for output by TYPE.

1.5.12 SIGN (v --)
If v is negative insert a '-' into the formatting string during <# …#> formatting.

1.5.13 #S (v -- 0)
Complete formatting of the number on the stack, leaving 0.

1.5.14 # (v – v%BASE)
Extract one digit using the current number base from the current number on the TOS and insert into the

format string.

1.5.15 HOLD (v --)
Insert the character into the format string.

1.5.16 <# (--)
Start a NULL terminated format string in the PAD buffer.

1.5.17 EXTRACT (v – v d)
Produce the least significant digit from v in the current number base. Reduce v by dividing by the current number base.

1.5.18 DIGIT (v – d)
Produce the ASCII character that represents the given value (up to base 36).

1.5.19 HANDLER (-- addr)
Return the address of the HANDLER user variable.

1.5.20 BASE (-- addr)
Return the address of the variable that holds the current number base.

1.5.21 >IN (--)
Return the current parse location in the input buffer.

1.5.22 PARSE (d – token length)
Parses a string out of the input buffer starting at >IN, delimited by d (or end of line). Returns the count and pointer to the string.

1.5.23 KEY (-- v)
Reads the next input character from the input. Waits until it is available. (See ?KEY).

1.5.24 EMIT (v --)
Print the value on the top of the stack as a character.

1.5.25 SPACE (--)
Output a single space on the console.

1.5.26 BL (-- v)
Push the ASCII value for a blank on the stack.

1.5.27 CR (--)
Output a newline sequence on the console.

1.5.28 .OK (--)
Print “OK” on the console with a newline sequence.

1.6 Control Structure Words

1.6.1 J (-- v)
Return the outer loop index in a nested DO LOOP.

1.6.2 I (-- v)
Return the loop index of the most inner DO LOOP.

1.6.3 NEXT (--)
Implement the end of a FOR NEXT loop.

1.6.4 FOR (v --)
Compiles a FOR loop. At runtime expects the number of times to loop on the TOS.

1.6.5 ELSE (--)
Starts the ELSE clause of an IF ELSE THEN compound statement. The statements from ELSE to THEN are executed if the IF condition is false.

1.6.6 THEN (--)
Close and IF as part of an IF THEN or IF THEN ELSE compound statements. Unconditional execution continues with the next statement after the THEN.

1.6.7 IF (T|F --)
If the value on the TOS is true execute the true part of an IF ELSE THEN or IF THEN statement.

1.6.8 REPEAT (--)
Return to the opening BEGIN statement in a BEGIN WHILE REPEAT loop.

1.6.9 WHILE (v --)
If the condition is false, branch out of a BEGIN WHILE REPEAT to the statement after the REPEAT. If the condition is true continue to execute the statements up to the repeat and branch back to the BEGIN.

1.6.10 AGAIN (--)
Close an infinite BEGIN AGAIN loop.

1.6.11 BEGIN (--)
Start a BEGIN WHILE REPEAT or BEGIN AGAIN loop.

1.6.12 LOOP (--)
Terminate a DO LOOP structure.

1.6.13 DO (TOP FIRST --)
Begin a DO LOOP structure.

1.7 Dictionary Words

1.7.1 STATUS (--)
Prints out the current usage of wordlist memory, stacks, etc.

1.7.2 HELP (--)
Reads the next word from the input and if help has been defined for that word, prints out the help.

1.7.3 ; (--)
Closes a word definitions.

1.7.4 : (--)
Expects a token to follow. Starts the definition of a new word.

1.7.5 ' (-- addr)
Finds the address of the next word in the input. Returns the code address that could be used by execute.

1.7.6 GREP (--)
Reads the next word from the input, then finds all words in all wordlists that contain this substring. Useful for finding words that you can only remember part of the name of.

1.7.7 WORDS (--)
List all the words in all the wordlists.

1.7.8 FORGET (--)
Truncate the current wordlist back to the word before the one named following forget. Use to remove temporary words and free up space.

1.7.9 NAME>TEXT (waddr -- addr)

Given a pointer to a forth word object, compute the address of the text field containing the name of the word.

1.7.10 >NAME (addr – addr)
Given the code address of a word (that returned by ‘), return the pointer to the word object itself.

1.7.11 NAME> (addr – addr)
Given the word object pointer, return the code pointer for the word.

1.7.12 NAME?W (addr – string 0 OR nameptr wordlistptr)
Given a pointer to counted string, search all wordlists (except the DEFINES wordlist) and if found return the wordlist and word object pointers.

1.7.13 NAME? (addr – strin 0 OR index flag)
Given a pointer to a counted string, search all wordlists (except the DEFINES wordlist) and if found return a flag and the index into the wordlist for the found word.

1.7.14 find (string wordlist – string 0 OR codeaddr TRUE)
Searches a wordlist for a counted string. If found return the execution address and true.

1.7.15 LOUD (--)
Print all words in all wordlists when doing WORDS.

1.7.16 QUIET (--)
Suppress common words in the WORDS output.

1.7.17 SOME-WORDS (waddr --)
Print all the words in the given wordlist.

1.7.18 RESTRICT (--)
Restrict the useable words to those only in the current compilation wordlist.

1.7.19 ADD-TO-ORDER (wordlist --)
Add this wordlist to the group of wordlists searched for words.

1.7.20 PREVIOUS (--)
Remove the most recently added wordlist from the list of wordlists.

1.7.21 ORDER (--)
Prints out the search (context) wordlists and the current wordlist for compilation.

1.7.22 ONLY (--)
Set the context to only the core words.

1.7.23 FORTH (--)
Replace the last context wordlist with the forth wordlist.

1.7.24 ALSO (--)
Duplicate the top wordlist in the context.

1.7.25 SET-ORDER (--)
Not implemented

1.7.26 SET-CURRENT (wordlist --)
Make the given wordlist the current compilation wordlist.

1.7.27 SEARCH-WORDLIST (--)
Not implemented.

1.7.28 GET-ORDER (-- w0 w1.. n)
Push the addresses of the current wordlists and the count of them.

1.7.29 GET-CURRENT (-- wordlist)
Return the current compilation wordlist.

1.7.30 FORTH-WORDLIST (-- wordlist)
Return the address of the main (non-core) wordlist.

1.7.31 DEFINITIONS (--)
Make the last wordlist the current wordlist also.

1.7.32 FREELIST (addr --)
Release a wordlist back to the OS.

1.7.33 WORDLIST(size namecount --)
Make a wordlist of the name following. Allocate a block of memory the size indicated and format the wordlist for the given number of word entries.

1.8 Miscellaneous Words

1.8.1 @EXECUTE (addr --)
Given the address of the address of a forth word, execute it. Used for deferred execution.

1.8.2 EXECUTE (addr --)
Given the address of a forth word (one given by ‘) execute the word.

1.8.3 // (--)
Ignore the remainder of this line of input

1.8.4 \ (--)
Ignore the remainder of this line of input.

1.8.5 ((--)
Ignore everything up to the next) found on this line.

1.8.6 SEE (--)
Decompile the word following SEE.

1.8.7 IMMEDIATE (--)
Mark the last word compiled as immediate.

1.8.8 COMPILE (--)
While compiled into a word, when the word executes, compile the word following compile into the current compile. Used to make words that do things at runtime.

1.8.9 $INTERPRET (--)
Part of the forth interpreter.

1.8.10 $COMPILE (--)
Part of the forth interpreter.

1.8.11 ABORT" (--)
Used by CATCH and THROW mechanism.

1.8.12 abort" (--)
Runtime part of ABORT”

1.8.13 THROW (v --)
Throw an exception to be caught by CATCH.

1.8.14 CATCH (caddr --)
Attempt to execute a word, allow for THROW to abort abnormally but gracefully.

1.8.15 QUIT (--)
The actual forth interpreter.

1.8.16 EVAL (--)
Part of the interpreter.

1.8.17 'EVAL (--)
Part of the interpreter.

1.8.18 TMP (--)
Return the address of a string storage area that is not overwritten by terminal input.

1.8.19 HUH (--)
Print “HUH?” and an error code.

1.8.20 USER (v --)
Return the address of the given user variable.

1.8.21 DELAY (v --)
Kill some time (system dependent).

1.8.22 ADD_EXTERN (--)
Part of the interpreter startup. Hook in external C functions.

1.9 Compiler Words

1.9.1 AHEAD

1.9.2 LITERAL

1.9.3 [']

1.9.4 [COMPILE]

1.9.5]

1.9.6 [

1.9.7 SAME?

1.9.8 PACK$

1.9.9 HERE

1.9.10 TOKEN

1.9.11 DEFINE?

1.9.12 NUMBER?

1.9.13 QUERY

1.9.14 :,

1.9.15 ,

1.9.16 $,n

1.9.17 STRING

1.9.18 CONSTANT,

1.9.19 VARIABLE,