Hitchhiker's Guide to FlashForth on PIC18 Microcontrollers

Interpreter

The outer interpreter looks for words and numbers delimited by whitespace. Everything is interpreted as a word or a number. Numbers are pushed onto the stack. Words are looked up and acted upon. Names of words are limited to 15 characters.

Data and the stack

The data stack is directly accessible and has 48 16-bit cells for holding numerical values. Functions get their arguments from the stack and leave their results there as well. There is also a return address stack (R:) that can be used for temporary storage.

Notation

`n, n1, n2, n3`	Single-cell integers (16-bit).
`u`	Unsigned integer (16-bit).
`x, x1, x2, x3`	Single-cell item (16-bit).
`c`	Character value (8-bit).
`d`	Double-cell value (32-bit).
`ud`	Unsigned double-cell value (32-bit).
`f`	Boolean flag: 0 is false, -1 is true.
`addr, addr1, addr2`	16-bit addresses.
`a-addr`	cell-aligned address.
`c-addr`	character or byte address.

Numbers and values

`2` Leave integer two onto the stack. `( -- 2 )`

`#255` Leave decimal 255 onto the stack. `( -- 255 )`

`%11` Leave integer three onto the stack. `( -- 3 )`

`$10` Leave integer sixteen onto the stack. `( -- 16 )`

`23.` Leave double number on the stack. `( -- 23 0 )`

`decimal` Set number format to base 10. `( -- )`

`hex` Set number format to hexadecimal. `( -- )`

`bin` Set number format to binary. `( -- )`

`s>d` Sign extend single to double number. `( n -- d )`
Since double numbers have the most significant bits in the cell above the least significant bits, you can just `drop` the top cell to recover the single number, provided that the value is not too large to fit in a single cell.

Displaying data

`.` Display a number. `( n -- )`

`u.` Display u unsigned. `( u -- )`

`u.r` Display u with field width n, $0 < n < 256$. `( u n -- )`

`d.` Display double number. `( d -- )`

`ud.` Display unsigned double number. `( ud -- )`

`.s` Display stack content (nondestructively).

`.st` Emit status string for base, current data section, and display the stack contents. `( -- )`

`dump` Display memory from address, for u bytes. `( addr u -- )`

Stack manipulation

`dup` Duplicate top item. `( x -- x x )`

`?dup` Duplicate top item if nonzero. `( x -- 0 | x x )`

`swap` Swap top two items. `( x1 x2 -- x2 x1 )`

`over` Copy second item to top. `( x1 x2 -- x1 x2 x1 )`

`drop` Discard top item. `( x -- )`

`nip` Remove x1 from the stack. `( x1 x2 -- x2 )`

`rot` Rotate top three items. `( x1 x2 x3 -- x2 x3 x1 )`

`tuck` Insert x2 below x1 in the stack. `( x1 x2 -- x2 x1 x2 )`

`pick` Duplicate the u-th item on top.
`( xu ... x0 u -- xu ... x0 xu )`

`2dup` Duplicate top item. `( d -- d d )`

`2swap` Swap top two double-cell items. `( d1 d2 -- d2 d1 )`

`2over` Copy second item to top. `( d1 d2 -- d1 d2 d1 )`

`2drop` Discard top item. `( d -- )`

`>r` Send to return stack. `( n -- ) ( R: -- n )`

`r>` Take from return stack. `( -- n ) ( R: n -- )`

`r@` Copy top item of return stack. `( -- n ) ( R: n -- n )`

`rdrop` Discard top item fro return stack. `( -- ) ( R: n -- )`

`sp@` Leave data stack pointer. `( -- addr )`

`sp!` Set the data stack pointer to address. `( addr -- )`

Operators

Arithmetic

Some of these words come from `core.txt`.

`+` Add. `( n1 n2 -- n1+n2 )` sum

`-` Subtract. `( n1 n2 -- n1-n2 )` difference

`*` Multiply. `( n1 n2 -- n1*n2 )` product

`/` Divide. `( n1 n2 -- n1/n2 )` quotient

`mod` Divide. `( n1 n2 -- n.rem )` remainder

`/mod` Divide. `( n1 n2 -- n.rem n.quot )`

`*/` Scale. `( n1 n2 n3 -- n1*n2/n3 )`
Uses 32-bit intermediate result.

`*/mod` Scale with remainder. `( n1 n2 n3 -- n.rem n.quot )`
Uses 32-bit intermediate result.

`u*/mod` Unsigned Scale u1*u2/u3 `( u1 u2 u3 -- u.rem u.quot )`
Uses 32-bit intermediate result.

`um*` Unsigned 16x16 to 32 bit multiply. `( u1 u2 -- ud )`

`ud*` Unsigned 32x16 to 32-bit multiply. `( ud u -- ud )`

`u/` Unsigned 16/16 to 16-bit division. `( u1 u2 -- u2/u1 )`

`u/mod` Unsigned division. `( u1 u2 -- u.rem u.quot )`
16-bit/16-bit to 16-bit

`um/mod` Unsigned division. `( ud u1 -- u.rem u.quot )`
32-bit/16-bit to 16-bit

`ud/mod` Unsigned division. `( ud u1 -- u.rem ud.quot )`
32-bit/16-bit to 32-bit

`fm/mod` Floored division. `( d n -- n.rem n.quot )`

`sm/rem` Symmetric division. `( d n -- n.rem n.quot )`
32-bit/16-bit to 16-bit.

`d+` Add double numbers. `( d1 d2 -- d1+d2 )`

`d-` Subtract double numbers. `( d1 d2 -- d1-d2 )`

`m+` Add double word to cell. `( d1 n -- d2 )`

`m*` Signed 16*16 to 32-bit multiply. `( n n -- d )`

`1+` Add one. `( n -- n1 )`

`1-` Subtract one. `( n -- n1 )`

`2+` Add two. `( n -- n1 )`

`2-` Subtract 2 from n. `( n -- n1 )`

`2*` Shift left by one bit. `( u -- u1 )`

`2/` Shift right by one bit. `( u -- u1 )`

`d2*` Multiply by 2. `( d -- d )`

`d2/` Divide by 2. `( d -- d )`

`lshift` Left shift by u bits. `( x1 u -- x2 )`

`rshift` Right shift by u bits. `( x1 u -- x2 )`

`negate` Negate n. `( n -- -n )`

`?negate` Negate n1 if n2 is negative. `( n1 n2 -- n3 )`

`dnegate` Negate double number. `( d -- d )`

`abs` Absolute value. `( n -- n )`

`dabs` Absolute value. `( d -- d )`

`min` Leave minimum. `( n1 n2 -- n )`

`max` Leave maximum. `( n1 n2 -- n )`

`umin` Unsigned minimum. `( u1 u2 -- u )`

`umax` Unsigned maximum. `( u1 u2 -- u )`

Relational

`=` Leave true if x1 x2 are equal. `( x1 x2 -- f )`

`<>` Leave true if x1 x2 are not equal. `( x1 x2 -- f )`

`<` Leave true if n1 less than n2. `( n1 n2 -- f )`

`>` Leave true if n1 greater than n2. `( n1 n2 -- f )`

`0=` Leave true if n is zero. `( n -- f )`
Inverts logical value.

`0<` Leave true if n is negative. `( n -- f )`

`within` Leave true if xl <= x < xh. `( x xl xh -- f )`

`d0=` Leave true if d is zero. `( d -- f )`

`d0<` Leave true if d is negative. `( d -- f )`

`d<` Leave true if d1 < d2. `( d1 d2 -- f )`

`d>` Leave true if d1 > d2. `( d1 d2 -- f )`

Bitwise

`invert` Ones complement. `( x -- x )`

`dinvert` Invert double number. `( du -- du )`

`and` Bitwise and. `( x1 x2 -- x )`

`or` Bitwise or. `( x1 x2 -- x )`

`xor` Bitwise exclusive-or. `( x -- x )`

`mset` Set bits in file register with mask c. `( c addr -- )`

`mclr` Clear bits in file register with mask c. `( c addr -- )`

`mtst` AND file register byte with mask c. `( c addr -- x )`

Interaction with the operator

Interaction with the user is via the serial port, typically UART1. Settings are 38400 baud, 8N1, using Xon/Xoff handshaking.

`tx0` Send a character via the USB UART. `( c -- )`

`rx0` Receive a character from the USB UART. `( -- c )`
Use hardware flow control.

`tx1` Send character to UART1. `( c -- )`
Buffered via a 32 byte interrupt driven queue.
`rx1` Receive a character from UART1. `( -- c )`
Has a 64-byte interrupt buffer.
`rx1?` Leave the number of characters in queue. `( -- n )`
`u1-` Disable flow control for operator interface. `( -- )`
`u1+` Enable flow control for operator interface, default. `( -- )`
`emit` Emit c to the serial port FIFO. `( c -- )`
FIFO is 46 chars. Executes pause.
`space` Emit one space character. `( -- )`
`spaces` Emit n space characters. `( n -- )`
`cr` Emit carriage-return, line-feed. `( -- )`
`key` Get a character from the serial port FIFO.
Execute pause until a character is available. `( -- c )`
`key?` Leave true if character is waiting
in the serial port FIFO. `( -- f )`

Other Hardware

`cwd` Clear the WatchDog counter. `( -- )`
`ei` Enable interrupts. `( -- )`
`di` Disable interrupts. `( -- )`
`ms` Pause for +n milliseconds. `( +n -- )`
`ticks` System ticks, 0–ffff milliseconds. `( -- u )`

Constants, variables and memory

Memory map

All operations are restricted to 64kB address space that is divided into three spaces:
`$0000` – `$ebff` Flash
`$ec00` – `$efff` EEPROM
`$f000` – `$ffff` SRAM
Using the default values in `p18f-main.cfg` for the UART version of FF, SRAM is further subdivided as:
`$f000` – `$f03f` 64-byte flash write buffer
`$f040` – `$f05f` 32-byte used internally by FF
`$f060` – `$f06f` 16-byte interrupt parameter stack
`$f070` – `$f093` 36-byte RX buffer (32) and TX buffer (4)
`$f094` – `$f09d` 10-byte mirror of turnkey, dp, latest
`$f09e` – `$f09f` 2-byte interrupt vector
`$f0a0` – `$f0a1` 2-byte user pointer
`$f0a2` – `$f1b1` 272-byte user area for operator task
A total of 434 bytes is dedicated to the FF system.
`$f1b2` – `$ff5f` Free for application use, up to `RAM_HI` in `p18fxxxx.cfg`.
`$ff60` – `$ffff` Special function registers
For the PIC18F2520 MCU, `RAM_HI` is `$f5e0`, leaving 1070 bytes for application use.

Context

`ram` Set address context to SRAM.
`eeprom` Set address context to EEPROM.
`flash` Set address context to Flash.
`fl-` Disable writes to Flash, EEPROM.
`fl+` Enable writes to Flash, EEPROM, default.
`lock` Disable writes to Flash, EEPROM.

Defining

`con` *name* Define new constant, inline code. `( n -- )`
`constant` *name* Define new constant, docreate. `( n -- )`
`variable` *varname* Define variable in address context. `( -- )`
`value` *valname* Define value. `( n -- )`
`to` *valname* Assign new value to *valname*. `( n -- )`
`2con` *name* Define double constant. `( x x -- )`
`2variable` *name* Define double variable. `( -- )`

Accessing

varname Leave address of variable on stack. `( -- addr )`
valname Leave value on stack. `( -- n )`
`!` Store x to address. `( x a-addr -- )`
`@` Fetch from address. `( a-addr -- x )`
`c!` Store character to address. `( c addr -- )`
`c@` Fetch character from address. `( addr -- c )`
`c@+` Fetch char, increment address.
`( addr1 -- addr2 c )`
`+!` Add n to cell at address. `( n addr -- )`
`-@` Fetch from addr and decrement addr by 2.
`( addr1 -- addr2 x )`

Examples

```
ram                     Set SRAM context for variables and
                        values. Be careful not to accidentally
                        define variables in EEPROM or Flash
                        memory. That memory wears quickly
                        with multiple writes.
$ff81 con portb         Define constant in Flash.
3 value xx              Define value in SRAM.
variable yy             Define variable in SRAM.
6 yy !                  Store 6 in variable yy.
eeprom 5 value zz ram   Define value in EEPROM.
xx yy zz portb yy @     Leaves 3 f172 5 ff81 6
warm                    Warm restart clears SRAM data.
xx yy zz portb yy @     Leaves 0 f172 5 ff81 0
4 to xx                 Sets new value.
xx yy zz portb yy @     Leaves 4 f172 5 ff81 0
```

Converting between cells, chars

`cells` Convert cells to address units. `( u -- u )`
`chars` Convert chars to address units. `( u -- u )`
`char+` Add one to address. `( addr1 -- addr2 )`
`cell+` Add size of cell to address. `( addr1 -- addr2 )`
`aligned` Align address to a cell boundary. `( addr -- a-addr )`

Memory operations

Some of these words come from `core.txt`.
`cmove` Move u bytes from address-1 to address-2.
`( addr1 addr2 u -- )`
Copy proceeds from low addr to high address.
`fill` Fill u bytes with c starting at address.
`( addr u c -- )`
`erase` Fill u bytes with 0 starting at address.
`( addr u -- )`
`blanks` Fill u bytes with spaces starting at address.
`( addr u -- )`

The P register

The P register can be used as a variable or as a pointer. It can be used in conjunction with `for..next` or at any other time.
`!p` Store address to P(ointer) register. `( addr -- )`
`@p` Fetch the P register to the stack. `( -- addr )`
`!p>r` Push contents of P to return stack and
store new address to P. `( addr -- ) ( R: -- addr )`
`r>p` Pop from return stack to P register. `( R: addr -- )`
`p+` Increment P register by one. `( -- )`
`p2+` Add 2 to P register. `( -- )`
`p++` Add n to the p register. `( n -- )`
`p!` Store x to the location pointed to
by the p register. `( x -- )`
`pc!` Store c to the location pointed to
by the p register. `( c -- )`
`p@` Fetch the cell pointed to
by the p register. `( -- x )`
`pc@` Fetch the char pointed to
by the p register. `( -- c )`
In a definition `!p>r` and `r>p` should always be used to allow proper nesting of words.

Predefined constants

`cell` Size of one cell in characters. `( -- n )`
`true` Boolean true value. `( -- -1 )`
`false` Boolean false value. `( -- 0 )`
`bl` ASCII space. `( -- c )`
`Fcy` Leave the cpu instruction-cycle frequency in kHz. `( -- u )`
`ti#` Size of the terminal input buffer. `( -- u )`

Predefined variables

`base` Variable containing number base. `( -- a-addr )`
`irq` Interrupt vector (SRAM variable). `( -- a-addr )`
Always disable user interrupts and clear related interrupt enable bits before zeroing interrupt vector.
`di false to irq ei`
`turnkey` Vector for user start-up word. `( -- a-addr )`
EEPROM value mirrored in SRAM.
`prompt` Deferred execution vector for the info displayed by quit. `( -- a-addr )`
`'emit` EMIT vector. Default is TX1. `( -- a-addr )`
`'key` KEY vector. Default is RX1. `( -- a-addr )`
`'key?` KEY? vector. Default is RX1. `( -- a-addr )`
`'source` Current input source. `( -- a-addr )`
`s0` Variable for start of data stack. `( -- a-addr )`
`rcnt` Number of saved return stack cells. `( -- a-addr )`
`tib` Address of the terminal input buffer. `( -- a-addr )`
`tiu` Terminal input buffer pointer. `( -- a-addr )`
`>in` Variable containing the offset, in characters, from the start of `tib` to the current parse area. `( -- a-addr )`

`pad` Address of the temporary area for strings. `( -- addr )`
`: pad tib ti# + ;`
Each task has its own pad but has zero default size.
If needed the user must allocate it separately
with allot for each task.

`dp` Leave the address of the current data section
dictionary pointer. `( -- addr )`
EEPROM variable mirrored in RAM.

`hp` Hold pointer for formatted numeric output. `( -- a-addr )`

Characters

`digit?` Convert char to a digit according to base.
`( c -- n )`

`>digit` Convert n to ascii character value. `( n -- c )`

`char` *char* Parse a character and leave ASCII value. `( -- n )`
For example: `char A ( -- 65 )`

`[char]` *char* Compile inline ASCII character. `( -- )`

Strings

Some of these words come from `core.txt`.

`s"` *text*`"` Compile string into flash. `( -- )`
At run time, leaves address and length.
`( -- addr u )`

`."` *text*`"` Compile string to print into flash.
`( -- )`

`place` Place string from a1 to a2
as a counted string. `( addr1 u addr2 -- )`

`count` Leave the address and length of text portion
of a counted string `( addr1 -- addr2 n )`

`n=` Compare strings in RAM(a) and flash(nfa).
Leave true if strings match, `n < 16`.
`( addr nfa u -- f )`

`/string` Trim string. `( addr u n -- addr+n u-n )`

`>number` Convert string to a number.
`( 0 0 addr1 u1 -- ud.l ud.h addr2 u2 )`

`number?` Convert string to a number and flag.
`( addr1 -- addr2 0 | n 1 | d.l d.h 2 )`
Prefix: `#` decimal, `$` hexadecimal, `%` binary.

`type` Type line to terminal, `u < $100. ( addr u -- )`

`accept` Get line from the terminal. `( c-addr +n1 -- +n2 )`
At most n1 characters are accepted, until the line
is terminated with a carriage return.

`source` Leave address and length of input buffer.
`( -- c-addr u )`

`evaluate` Interpret a string in SRAM. `( addr n -- )`

Pictured numeric output

Formatted string representing an unigned double-precision integer is constructed in the end of `tib`.

`<#` Begin conversion to formatted string. `( -- )`

`#` Convert 1 digit to formatted string. `( ud1 -- ud2 )`

`#s` Convert remaining digits. `( ud1 -- ud2 )`
Note that `ud2` will be zero.

`hold` Append char to formatted string. `( c -- )`

`sign` Add minus sign to formatted string, if `n<0`. `( n -- )`

`#>` End conversion, leave address and count
of formatted string. `( ud1 -- c-addr u )`

For example, the following:
`-1 34. <# # # #s rot sign #> type`
results in `-034 ok`

Defining functions

Colon definitions

`:` Begin colon definition. `( -- )`

`;` End colon definition. `( -- )`

`[` Enter interpreter state. `( -- )`

`]` Enter compilation state. `( -- )`

`[i` Enter Forth interrupt context. `( -- )`

`i]` Enter compilation state. `( -- )`

`;i` End an interrupt word. `( -- )`

`literal` Compile value on stack at compile time.
At run time, leave value on stack. `( -- x )`

`inline` *name* Inline the following word. `( -- )`

`inlined` Mark the last compiled word as inlined. `( -- )`

`immediate` Mark latest definition as immediate. `( -- )`

`postpone` *name* Postpone action of immediate word. `( -- )`

`see` *name* Show definition. Load `see.txt`.

Comments

`(` *comment text*`)` Inline comment.

`\` *comment text* Skip rest of line.

Examples

```
: square ( n -- n**2 )      Example with stack comment.
  dup * ;                   ...body of definition.
: poke0  ( -- )             Example of using assembler.
  [ $f8a 0 a, bsf, ] ;
```

Flow control

Structured flow control

`if` *xxx* `else` *yyy* `then` Conditional execution. `( f -- )`

`begin` *xxx* `again` Infinite loop. `( -- )`

`begin` *xxx cond* `until` Loop until *cond* is true. `( -- )`

`begin` *xxx cond* `while` Loop while *cond* is true. `( -- )`
yyy `repeat` *yyy* is not executed on the last iteration.

`for` *xxx* `next` Loop u times. `( u -- )`
`r@` gets the loop counter u-1 ... 0

`leave` Sets `r@` to zero so that we leave
a for loop when next is encountered.
`( -- )`

Unstructured flow control

`exit` Exit from a word. `( -- )`
If exiting from within a for loop,
we must drop the loop count with `rdrop`.

`?abort` If flag is false, print message
and abort. `( f addr u -- )`

`?abort?` If flag is false, output ? and abort. `( f -- )`

`abort" xxx"` if flag, type out last word executed,
followed by text xxx. `( f -- )`

`quit` Interpret from keyboard. `( -- )`

`cold` Make a cold start.
Reset all dictionary pointers.

`warm` Make a warm start.
Note that irq vector is cleared.

Function pointers (vectors)

`'` *name* Search for *name* and leave its
execution token (address). `( -- addr )`

`[']` *name* Search for *name* and compile
it's execution token. `( -- )`

`execute` Execute word at address. `( addr -- )`
The actual stack effect will depend on
the word executed.

`@ex` Fetch vector from addr and execute.
`( addr -- )`

`defer` *vec-name* Define a deferred execution vector. `( -- )`

`is` *vec-name* Store execution token in *vec-name*.
`( addr -- )`

vec-name Execute the word whose execution token
is stored in *vec-name*'s data space.

`' my-app is turnkey` Autostart my-app.

`false is turnkey` Disable turnkey application.

Interrupt example

```
ram variable icnt1          ...from FF source.
: irq_forth                 It's a Forth colon definition
  [i                        ...in the Forth interrupt context.
    icnt1 @ 1+
    icnt1 !
  ]i
;i
' irq_forth is irq          Set the user interrupt vector.

: init                      Alternatively, compile a word
  ['] irq_forth is irq      ...so that we can install the
;                           ...interrupt service function
' init is turnkey           ...at every warm start.
```

Multitasking

Load the words for multitasking from `task.txt`.

`task` Define a new task in flash memory space
`( tibsize stacksize rsize addsize -- )`
Use ram xxx allot to leave space for the PAD
of the prevously defined task.
The OPERATOR task does not use PAD.

`tinit` Initialise a user area and link it
to the task loop. `( taskloop-addr task-addr -- )`
Note that this may only be executed from
the operator task.

`run` Makes a task run by inserting it after operator
in the round-robin linked list. `( task-addr -- )`
May only be executed from the operator task.

`end` Remove a task from the task list. `( task-addr -- )`
May only be executed from the operator task.

`single` End all tasks except the operator task. `( -- )`
Removes all tasks from the task list.
May only be executed from the operator task.

`tasks` List all running tasks. `( -- )`

`pause` Switch to the next task in the
round robin task list. `( -- )`

`his` Access user variables of other task.
`( task.addr vvar.addr -- addr )`

`load` Leave the CPU load on the stack. `( -- n )`
Load is percentage of time that the CPU is busy.
Updated every 256 milliseconds.

`busy` CPU idle mode not allowed. `( -- )`

`idle` CPU idle is allowed. `( -- )`

`operator` Leave the address of the operator task. `( -- )`

`ulink` Link to next task. `( -- addr )`

Defining compound data objects

`create` *name* Create a word definition and store
the current data section pointer.

`does>` Define the runtime action of a created word.

`allot` Advance the current data section dictionary
pointer by u bytes. `( u -- )`

`,` Append x to the current data section. `( x -- )`

`c,` Append c to the current data section. `( c -- )`

`cf,` Compile xt into the flash dictionary. `( addr -- )`

`c>n` Convert code field addr to name field addr.
`( addr1 -- addr2 )`

`n>c` Convert name field addr to code field addr.
`( addr1 -- addr2 )`

`," xxx"` Append a string at HERE. `( -- )`

`here` Leave the current data section dictionary
pointer. `( -- addr )`

`align` Align the current data section dictionary
pointer to cell boundary. `( -- )`

Examples

```
ram                             Example
create my-array 20 allot        ...of creating an array,
my-array 20 $ff fill            ...filling it with 1s, and
my-array 20 dump                ...displaying its content.
create my-cell-array
    100 , 340 , 5 ,               Initialised cell array.
my-cell-array 2 cells + @       Should leave 5. ( -- x )
create my-byte-array
    18 c, 21 c, 255 c,            Initialised byte array.
my-byte-array 2 chars + c@      Should leave 255. ( -- c )
: mk-byte-array                 Defining word ( n -- )
    create allot                ...to make byte array objects
    does> + ;                   ...as shown in FF user's guide.
10 mk-byte-array my-bytes       Creates an array object
                                my-bytes ( n -- addr ).
18 0 my-bytes c!                Sets an element
21 1 my-bytes c!                ...and another.
255 2 my-bytes c!
2 my-bytes c@                   Should leave 255.
: mk-cell-array                 Defining word ( n -- )
    create cells allot          ...to make cell array objects.
    does> swap cells + ;
5 mk-cell-array my-cells        Creates an array object
                                my-cells ( n -- addr ).
3000 0 my-cells !               Sets an element
45000 1 my-cells !              ...and another.
63000 2 my-cells !
1 my-cells @ .                  Should print 45000
```

Dictionary manipulation

`marker -my-mark` Mark the dictionary state with -my-mark.

`-my-mark` Discard back to -my-mark.

`find` *name* Find name in dictionary. `( -- n )`
Leave 1 immediate, -1 normal, 0 not found.

`forget` *name* Forget word *name*.

`empty` Reset all dictionary pointers. `( -- )`

`words` List words in dictionary. `( -- )`

Assembler words

To use many of the words below, load the text file `asm.txt`. In the stack-effect notaion, f is a file register address, d is the result destination, a is the access bank modifier, and k is a literal value.

Destination and access modifiers

`w,` Destination WREG `( -- 0 )`

`f,` Destination file `( -- 1 )`

`a,` Access bank `( -- 0 )`

`b,` Use bank-select register `( -- 1 )`

Byte-oriented file register operations

`addwf,` Add WREG and f. `( f d a -- )`

`addwfc,` Add WREG and carry bit to f. `( f d a -- )`

`andwf,` AND WREG with f. `( f d a -- )`

`clrf,` Clear f. `( f a -- )`

`comf,` Complement f. `( f d a -- )`

`cpfseq,` Compare f with WREG, skip if equal. `( f a -- )`

`cpfsgt,` Compare f with WREG, skip if greater than. `( f a -- )`

`cpfslt,` Compare f with WREG, skip if less than. `( f a -- )`

`decf,` Decrement f. `( f d a -- )`

`decfsz,` Decrement f, skip if zero. `( f d a -- )`

`dcfsnz,` Decrement f, skip if not zero. `( f d a -- )`

`incf,` Increment f. `( f d a -- )`

`incfsz,` Increment f, skip if zero. `( f d a -- )`

`infsnz,` Increment f, skip if not zero. `( f d a -- )`

`iorwf,` Inclusive OR WREG with f. `( f d a -- )`

`movf,` Move f. `( f d a -- )`

`movff,` Move fs to fd. `( fs fd -- )`

`movwf,` Move WREG to f. `( f a -- )`

`mulwf,` Multiply WREG with f. `( f a -- )`

`negf,` Negate f. `( f a -- )`

`rlcf,` Rotate left f, through carry. `( f d a -- )`

`rlncf,` Rotate left f, no carry. `( f d a -- )`

`rrcf,` Rotate right f, through carry. `( f d a -- )`

`rrncf,` Rotate right f, no carry. `( f d a -- )`

`setf,` Set f. `( f d a -- )`

`subfwb,` Subtract f from WREG, with borrow. `( f d a -- )`

`subwf,` Subtract WREG from f. `( f d a -- )`

`subwfb,` Subtract WREG from f, with borrow. `( f d a -- )`

`swapf,` Swap nibbles in f. `( f d a -- )`

`tstfsz,` Test f, skip if zero. `( f a -- )`

`xorwf,` Exclusive OR WREG with f. `( f d a -- )`

Bit-oriented file register operations

`bcf,` Bit clear f. `( f b a -- )`

`bsf,` Bit set f. `( f b a -- )`

`btfsc,` Bit test f, skip if clear. `( f b a -- )`

`btfss,` Bit test f, skip if set. `( f b a -- )`

`btg,` Bit toggle f. `( f b a -- )`

Literal operations

`addlw,` Add literal and WREG. `( k -- )`

`andlw,` AND literal with WREG. `( k -- )`

`iorlw,` Inclusive OR literal with WREG. `( k -- )`

`lfsr,` Move literal to FSRx. `( k f -- )`

`movlb,` Move literal to BSR. `( k -- )`

`movlw,` Move literal to WREG. `( k -- )`

`mullw,` Multiply literal with WREG. `( k -- )`

`sublw,` Subtract WREG from literal. `( k -- )`

`xorlw,` Exclusive OR literal with WREG. `( k -- )`

Data memory – program memory operations

`tblrd*,` Table read. `( -- )`

`tblrd*+,` Table read with post-increment. `( -- )`

`tblrd*-,` Table read with post-decrement. `( -- )`

`tblrd+*,` Table read with pre-increment. `( -- )`

`tblwt*,` Table write. `( -- )`

`tblwt*+,` Table write with post-increment. `( -- )`

`tblwt*-,` Table write with post-decrement. `( -- )`

`tblwt+*,` Table write with pre-increment. `( -- )`

Low-level flow control

`bra,`	Branch unconditionally. `( rel-addr -- )`
`call,`	Call subroutine. `( addr -- )`
`clrwdt,`	Clear watchdog timer. `( -- )`
`daw,`	Decimal adjust WREG. `( -- )`
`goto,`	Go to address. `( addr -- )`
`nop,`	No operation. `( -- )`
`pop,`	Pop top of return stack. `( -- )`
`push,`	Push top of return stack. `( -- )`
`rcall,`	Relative call. `( rel-addr -- )`
`reset,`	Software device reset. `( -- )`
`retfie,`	Return from interrupt enable. `( -- )`
`retlw,`	Return with literal in WREG. `( k -- )`
`return,`	Return from subroutine. `( -- )`
`sleep,`	Go into standby mode. `( -- )`

Structured flow control

`if,` *xxx* `else,` *yyy* `then,`	Conditional execution. `( cc -- )`
`begin,` *xxx* `again,`	Loop indefinitely. `( -- )`
`begin,` *xxx cc* `until,`	Loop until condion is true. `( -- )`

Conditions for structured flow control

`cc,`	test carry `( -- cc )`
`nc,`	test not carry `( -- cc )`
`mi,`	test negative `( -- cc )`
`pl,`	test not negative `( -- cc )`
`z,`	test zero `( -- cc )`
`nz,`	test not zero `( -- cc )`
`ov,`	test overflow `( -- cc )`
`nov,`	test not overflow `( -- cc )`
`not,`	invert condition `( cc -- not-cc )`

Extras

I^{2}C communications as master

Load these words from `i2c_base.txt`.

`i2cinit`	Initializes I^{2}C master mode, 100 kHz clock. `( -- )`
`i2cws`	Wake slave. Bit 0 is R/W bit. `( slave-addr -- )` The 7-bit I^{2}C address is in bits 7-1.
`i2c!`	Write one byte to I^{2}C bus and wait for `ACK`. `( c -- )`
`i2c@ak`	Read one byte and continue. `( -- c )`
`i2c@nak`	Read one last byte from the I^{2}C bus. `( -- c )`
`i2c-addr1`	Write 8-bit address to slave. `( addr slave-addr -- )`
`i2c-addr2`	Write 16-bit address to slave `( addr slave-addr -- )`

Lower-level words.

`ssen`	Assert start condition. `( -- )`
`srsen`	Assert repeated start condition. `( -- )`
`spen`	Generate a stop condition. `( -- )`
`srcen`	Set receive enable. `( -- )`
`snoack`	Send not-acknowledge. `( -- )`
`sack`	Send acknowledge bit. `( -- )`
`sspbuf!`	Write byte to `SSPBUF` and wait for transmission. `( c -- )`

This guide assembled by Peter Jacobs, School of Mechanical Engineering, The University of Queensland, May – 07-Jan-2013 as Report 2012/06.
It is a remix of material from the following sources:
FlashForth v3.8 source code and word list by Mikael Nordman http://flashforth.sourceforge.net/
EK Conklin and ED Rather *Forth Programmer's Handbook* 3rd Ed. 2007 FORTH, Inc.
L Brodie *Starting Forth* 2nd Ed., 1987 Prentice-Hall Software Series.
Robert B. Reese *Microprocessors from Assembly Language to C Using the PIC18Fxx2* Da Vinci Engineering Press, 2005.