# PIC 16F87x Forth programmer manual

Using PicForth 0.23
2 November 2002

**Samuel Tardieu**

# 1 Preamble

Microchip PIC 16F87x microcontrollers are very well suited for a number of tasks. However, the programmer is left with several choices to program them:

- Use Microchip MPLab IDE running on Microsoft Windows with the assembly programming language.
- Buy a third-party C compiler running on Microsoft Windows.
- Use the `gputils` package on a Unix system and program the PIC using the assembly language.
- Use the `sdcc` C compiler on a Unix system and program the PIC in C.
- Use the `PicForth` Forth compiler on a Unix system and program the PIC in Forth.

We do believe that the latest is a very pleasant solution for PIC development, as Forth is particularily suited to embedded systems, and Unix is more user-friendly for the developper.

**Warning:** this manual is a work-in-progress, and is in no way complete.

# 2 Introduction

## 2.1 What is that?

This program is a Forth compiler for the Microchip PIC 16F87x family.

## 2.2 Why this project?

I needed to write some code on a PIC to control a digital model railroad system using the DCC (Digital Control Command) protocol. However, writing it in assembly is error-prone and writing it in C is no fun as C compiled code typically needs a lot of space.

So I wrote this compiler, not for the purpose of writing a compiler, but as a tool to write my DCC engine.

## 2.3 State of the compiler

The compiler does not aim to be ANS Forth compliant. It has quite a few words already implemented, and I will implement more of them as needed. Of course, you are welcome to contribute some (see below for license information).

At this time, many words are missing from standard Forth. For example, I have no multiply operation as I have no use for it at this time and won't spend time to implement things I don't need (remember, Forth is a tool before anything else).

## 2.4 License

The compiler is released at the moment under the GNU General Public License version 2 (I intend to use the less restrictive BSD license in the future, but as it is based on gforth, I have to sort out those issues with gforth copyright holders).

However, the code produced by using this compiler is not tainted by the GPL license at all. You can do whatever you want with it, and I claim absolutely no right on the input or output of this compiler. I encourage to use it for whatever you want.

Note that I would really like people to send me their modifications (be they bug fixes or new features) so that I can incorporate them in the next release.

## 2.5 Why not use Mary?

Mary was a great inspiration source, I even kept some of the names from it. However, no code has been reused, as both Forth do not have the same goal.

## 2.6 Credits

I would like to thank the following people:

- Keith Wootten for his precious examples of how he uses a forth-ish assembler for the PIC and his inspiration for some control structures
- Francisco Rodrigo Escobedo Robles for his Mary PIC Forth compiler
- Herman Tamas for his suggestions for some word names
- Daniel Serpell for his superoptimizer (a program looking for the shortest possible sequences doing a particular job)

# 3 A very short Forth primer

## 3.1 Foreword

For a full introduction to the Forth programming language, please have a look at the appropriate section of the Open Directory (maintained by volunteers), at address http://dmoz.org/Computers/Programming/Languages/Forth/ . Only a small subset of the language will be presented here, sometimes overlooking details.

## 3.2 Words

The Forth programming language may look unusual to people used to other languages. First of all, the actions to execute are spelled one after each other. The sentence `init mainloop cleanup` will call, in turn, the word `init`, the word `mainloop` then the word `cleanup`.

To define a new word, the `:` defining word is used, while the `;` word ends the definition. The following code defines a new word `doit` which factors the three words used above:

```
: doit init mainloop cleanup ;
```

After it has been defined, the word `doit` can be called as other words by using its name. A Forth program is a collection of application-specific words. Each word, made of other words, will be used in turn to define new words, until the whole solution is described.

Words are similar to subprograms in more conventional programming languages. Any non-blank character can be part of a word name. For example, `\`, `^`, or `$` are legal characters in a word name, and can even be a word name by themselves.

## 3.3 Stack and arguments passing

In Forth, one does not use parenthesis to give arguments to called words. Instead, a stack is used, where the arguments can be pushed and where they can be popped from.

The word `+` pops two arguments from the top of the stack and pushes their sum. To push an integer to the top of the stack, one writes its value. The sentence `3 5 +` will push `3` on the stack, then `5`, and calls the word `+` which removes `3` and `5` and pushes `8`.

Some words do manipulate the stack explicitely. `dup` duplicates the element at the top of the stack, while `drop` removes it. `swap` exchanges the two top elements. The following word that we name `2*` (remember that this name is perfectly valid in Forth) does multiply the top of the stack by two, by adding it to itself:

```
: 2* dup + ;
```

The stack effect of a word is often written as a comment between parenthesis; those comments are ignored by the Forth compiler. The previously defined word could have been written:

```
: 2* ( n -- 2*n ) dup + ;
```

Elements on the stack are represented from left to right (top of the stack). For example, the `-` word which substract the top of the stack from the second element on the stack would have a stack comment looking like `( n1 n2 -- n1-n2 )`.

Let's assume that you want to multiply the top of the stack by four. You can define the `4*` word as:

```
: 4* ( n -- 4*n ) dup + dup + ;
```

But remember that you can define your own words from existing words. If you now need a word which multiplies the top of the stack by four, you can use your previously defined `2*` word:

```
: 4* ( n -- 4*n) 2* 2* ;
```

Definitions in Forth tend to be very short. The grouping of common parts in words is called **factoring**, and leads to very concise machine code.

## 3.4 Memory access

Two useful words allow you to access memory. `@` gets the content of the memory byte whose address is at the top of the stack and `!` stores, in the memory byte whose address is at the top of the stack, the following element.

The code below defines a word `mirror` which mirrors the content of port A into port B (we will later see more practical ways of defining some of the words seen here):

```
: porta 5 ;
: portb 6 ;
: mirror porta @ portb ! ;
```

## 3.5 Constant and variables

The defining word `constant` allows you to define named constants. Using this word, one can simplify the above example:

```
5 constant porta
6 constant portb
: mirror porta @ portb ! ;
```

The defining word `variable` reserves a byte in the PIC RAM and gives it a name:

```
5 constant porta
variable counter
: increment-counter counter @ 1 + counter ! ;
: counter-to-porta counter @ porta ! ;
```

## 3.6 Tests

Testing in Forth is done using a `if` construct, terminated by a `then`, with an optional `else`. Operators such as `<` or `=` can be used, and any non-null value is considered as true. The `abs` word changes the value on top of the stack to its absolute value (note that `abs` and `negate` are in fact already defined by PicForth):

```
: negate 0 swap - ;
: abs dup 0 < if negate then ;
```

The word `mirror` duplicates port A to port B or port C, depending on its argument; `0` for port B, anything else for port C (`porta`, `portb` and `portc` constant are already defined in PicForth):

```
: mirror ( n -- ) porta @ swap if portb ! else portc ! then ;
```

## 3.7 Loops

Several looping constructs are used in PicForth. The first of them is built upon `begin` and `again`, which here calls `do-one-thing` indefinitely:

```
: mainloop begin do-one-thing again ;
```

`while` and `repeat` can add a test in the loop and continue as long as the word `continue?` returns a non-null result:

```
: mainloop begin do-one-thing continue? while repeat ;
```

Note that `while` can be present anywhere between `begin` and `repeat`, letting you build elaborate constructs. Also, `until` allows you to wait for a condition. The following word calls `do-one-thing` until `end?` returns a non-null value:

```
: mainloop begin do-one-thing end? until ;
```

The last construct seen here is built around `v-for` and `v-next`. `v-for` takes a (non-included) high bound and a variable address on the stack. The following word `main` calls `do-one-thing` 10 times:

```
variable count
: main 10 count v-for do-one-thing count v-next ;
```

# 4 Our first PicForth program

## 4.1 The program itself

Our first PicForth program will generate a rectangle wave signal on port B0 as fast as possible:

```
0 pin-b i/o
: init i/o >output ;
: pulse i/o high i/o low ;
: mainloop begin pulse again ;
main : program init mainloop ;
```

## 4.2 Line by line explanation

The first line `0 pin-b i/o` defines a new word `i/o` which, when executed, will push two integers `6` (corresponding to portb) and `0` on the stack. This way, instead of writing `portb 0` to manipulate bit 0 of port B you can write `i/o`, which is shorter and lets you change it at only one place should you want to change which port is used.

The second line uses the PicForth word `>output` which sets the port whose address and bit are on the stack in output mode. This defines a new `init` word which initializes our port B0 as an output.

The third line creates a new word `pulse` which uses the PicForth words `high` and `low` to set a pin high or low. As a result, executing the `pulse` word will set the B0 pin high then low, this generating a pulse.

The fourth line defines a `mainloop` word which calls `pulse` endlessly, thus generating the rectangle wave signal we want.

The last line uses the PicForth word `main`. This word indicates to PicForth that the next word to be defined will be the one to call on reset. The word, called `program` here, calls `init` then `mainloop`. As `mainloop` never returns, the program runs until the end of time (which is usually considered quite a long time).

## 4.3 Generated assembly code

The generated code looks like:

```
0x0000  018A    clrf    0x0A
0x0001  280C    goto    0x00C   ; (init-picforth)
0x0002  0000    nop
        ; name: init
        ; max return-stack depth: 0
0x0003  1683    bsf     0x03,5
```

```
0x0004  1006    bcf     0x06,0
0x0005  1283    bcf     0x03,5
0x0006  0008    return
        ; name: pulse
        ; max return-stack depth: 0
0x0007  1406    bsf     0x06,0
0x0008  1006    bcf     0x06,0
0x0009  0008    return
        ; name: mainloop
        ; max return-stack depth: 1
0x000A  2007    call    0x007   ; pulse
0x000B  280A    goto    0x00A   ; mainloop (rs depth: 1)
        ; name: (init-picforth)
        ; max return-stack depth: 0
0x000C  3032    movlw   0x32
0x000D  0084    movwf   0x04
        ; name: program
        ; max return-stack depth: 1
0x000E  2003    call    0x003   ; init
0x000F  280A    goto    0x00A   ; mainloop (rs depth: 1)
```

## 4.4 An alternate solution

Of course, it is possible to write less factored code for such a simple task, and write instead:

```
0 pin-b i/o
main : program i/o >output begin i/o high i/o low repeat ;
```

In this case, it generates effectively a code which is a bit shorter:

```
0x0000  018A    clrf    0x0A
0x0001  2803    goto    0x003   ; (init-picforth)
0x0002  0000    nop
        ; name: (init-picforth)
        ; max return-stack depth: 0
0x0003  3032    movlw   0x32
0x0004  0084    movwf   0x04
        ; name: program
        ; max return-stack depth: 0
0x0005  1683    bsf     0x03,5
0x0006  1006    bcf     0x06,0
0x0007  1283    bcf     0x03,5
0x0008  1406    bsf     0x06,0
0x0009  1006    bcf     0x06,0
0x000A  2808    goto    0x008   ; program + 0x003
```

However, do not let this short example mislead you. While the code looks more efficient and shorter (and it is), this is generally not true for real-life programs. For example, in a bigger program it would be quite common to have to call `pulse` from other places.

## 4.5 Using inlined code

It is possible to use inlined code by surrounding the words you want to inline by the `macro` and `target` words:

```
0 pin-b i/o
macro
: init i/o >output ;
: pulse i/o high i/o low ;
: mainloop begin pulse again ;
target
main : program init mainloop ;
```

While this code is highly factored and easily maintainable, it generates the very same code as the less-factored version above.

# 5 Compiler documentation

## 5.1 Organisation

The stack is indexed by the only indirect register, fsr. The indf register automatically points to the top of stack.

The w register is used as a scratch. Attempts to use it to cache the top of stack proved to be inefficient, as we often need a scratch register.

## 5.2 Compiling

The compiler is hosted on gforth, a free software compiler for Unix systems. The command line to use to compile file '`foo.fs`' into '`foo.hex`', and getting a usable map into foo.map is:

```
gforth picforth.fs -e 'include foo.fs final-dump foo.hex map bye' | \
   sort -o foo.map
```

Of course, you should automate this in a Makefile, such as the one provided with the compiler.

If you install the GNU PIC utils (from http://gputils.sourceforge.net/), then you can read the assembled code by using `gpdasm`.

## 5.3 Code

The whole code space can be used. However, code generated in the first 2048 words is more efficient than the code generated in the following 2048 words; both are more efficient than the code generated for the remaining words. This is due to the PIC architecture which does not allow to see the code space as a flat zone.

## 5.4 Interactive mode

By executing

```
gforth picforth.fs -e 'host picquit'
```

(or `make interactive` from a Unix shell), you are dropped into an interactive mode, where you can use the following words to check your code:

```
see ( "name" -- )    Disassemble a word
map ( -- )           Print code memory map
dis ( -- )           Disassemble the whole code section
```

## 5.5 Literals

Hexadecimal literals should be prefixed by a dollar sign `$` to avoid confusion with existing constants (such as `c` for carry bit). This is a strong advice.

## 5.6 Default base

The default base is hexadecimal. Do not change it before including libraries bundled with the compiler, as they do expect hexadecimal mode.

## 5.7 Stack size

The default stack size is 16. If you use the multitasker included in '`multitasker.fs`' (see below), each task gets an additionnal 8 bytes of task-specific stack.

You can change the default stack size by using

```
set-stack-size ( n -- )
```

in interpretation mode before using `main`.

## 5.8 Shifting

`rlf-tos` and `rrf-tos` respectively shift the top-of-stack left or right, with the carry entering the byte and the outgoing bit entering the carry.

`lshift` and `rshift` used with a constant shift, and `2*` and `2/` do have the last exited bit in the carry.

`swapf-tos` will swap the upper and lower nibble of the top-of-stack.

## 5.9 Looping

There exists a `v-for`/`v-next` structure (v stands for variable):

```
v-for ( n addr -- )
  Initialize addr content with n.

v-next ( -- )
  Decrement addr content. If content is not zero,
  jump to v-for location.
```

The address has to be located in bank 0.

Also, the words `begin`, `again`, `while`, `until` and `repeat` are implemented.

## 5.10 Memory

You can choose the memory bank that will be used by the memory commands in interpretation mode by using the words `bank0`, `bank1`, `bank2` and `bank3` (check that it applies to your device first).

Those commands do affect the subsequent `create`, `variable`, `allot`, `,` and `here` commands. However, note that you can only access indirectly variables located in bank 0. Locations in other banks must be accessed using their static addresses.

You can define your own memory sections using the words `section`, `idata` and `udata`. No check will be made to ensure that those sections do not overlap.

## 5.11 Variables

Variables are not automatically initialized to zero, as this would waste too much code if it is not needed. If you want a variable explicitely initialized, use `create` and `,` such as in:

```
create attempts 3 ,
```

## 5.12 Tables

Tables can be created either in RAM (with run-time initialization, which is costly), in program flash memory or in the internal EEPROM.

The following words allow you to create tables:

```
table      ( "name" -- )        Start a RAM table
ftable     ( "name" -- )        Start a program flash table
eetable    ( "name" -- )        Start an EEPROM flash table
table>     ( "b1 .. bn" -- )    Add bytes b1 to bn in the table
end-table  ( -- )               End table declaration
```

The following code shows a table called `substitutions` and a `substitute` word which takes a byte in area `old-key` and sets it at the right place in area `new-key`, according to the `substitutions` table.

```
ftable substitutions
 table> 14 4 13 1 2 15 11 8 3 10 6 12 5 9 0 7
 table> 0 15 7 4 14 2 13 1 12 6 12 11 9 5 3 8
 table> 4 1 14 8 13 6 2 11 15 12 9 7 3 10 5 0
 table> 15 12 8 2 4 9 1 7 5 11 3 14 10 0 6 13
end-table

: substitute ( n -- ) dup old-key +  swap substitutions new-key + ! ;
```

## 5.13 Main program

A `main` word indicates that the next address is the main program. Use for example:

```
main : main-program ( -- )
  (do initialisations)
  (call mainloop)
;
```

## 5.14 Macros

You can switch to macro mode by using the `macro` word. You get back to target mode by using the `target` word.

## 5.15 Included files

You can include files using `include file` or `needs file` (which prevents from multiple inclusions to happen).

## 5.16 Assembler

There is a full prefix assembler included. Use `code` and `end-code` to define words written in assembler. `]asm` and `asm[` let you respectively switch to assembler mode and back during the compilation of a Forth word.

The `label:` defining word can be used to define a label that will then be used with `goto`. See the '`piceeprom.fs`' file for an example.

## 5.17 Interrupts

If you want to use interrupts, use

```
include picisr.fs
```

Two words do respectively save and restore the context around interrupt handling code:

```
isr-save ( -- )
isr-restore-return ( -- )
```

Also, the word `isr` is provided to notify that the next address is the isr handler.

For example, you can write an interrupt handler with:

```
isr : interrupt-handler ( -- )
  isr-save
     (interrupt handling code here)
  isr-restore-return
;
```

Do not forget that the return stack depth is only height. An interrupt can occur at any time unless you mask them or unset the GIE bit.

Two facility words that manipulate GIE are also provided:

```
enable-interrupts ( -- )
disable-interrupts ( -- )
```

You have to dispatch the interrupts and clear the interrupt bits manually before you return from the handler.

Versions that do nothing are provided in the default compiler. Useful versions are redefined when using '`picisr.fs`'.

Because of this, include '`picisr.fs`' as soon as possible, before other files and before using enable-interrupts and disable-interrupts. Other included files may fail to act properly if you don't.

## 5.18 Argument passing

In Forth, argument passing is done on the stack. However, if you want to transmit the top-of-stack argument in the w register (for example if a word typically takes a constant which is put on the stack just before calling it), you can use the defining word `::` instead of `:`. All calls will automatically use this convention.

If you want to return a value in the w register, you can use the word `>w` which loads the top-of-stack into the w register before every exit point. After calling a word which returns its result in the w register, you can call `w>` to put the w register value onto the stack.

## 5.19 Bit manipulation

To ease bit manipulation, the following words are defined for port p:

```
and!       ( n p -- )    logical and with n
/and!      ( n p -- )    logical and with ~n
/and       ( a b -- c )  logical and of a and ~b
or!        ( n p -- )    logical or with n
xor!       ( n p -- )    logical xor with n
invert!    ( p -- )      invert content
bit-set    ( p b -- )    set bit b of p (both have to be constants)
bit-clr    ( p b -- )    clear bit b of p (both have to be constants)
bit-toggle ( p b -- )    toggle bit b of p (both have to be constants)
bit-mask   ( p b -- m )  put 1<<b on stack
bit-set?   ( p b -- m )  put bit-mask (non-zero) on stack if bit b of
                         p is set, zero otherwise
bit-clr?   ( p b -- f )  true if bit b of p is clear
```

Six words help designate bit or port pins:

```
bit    ( n addr "name" -- )    ( Runtime: -- addr n )
pin-a  ( n "name" -- )         ( Runtime: -- porta n )
pin-b  ( n "name" -- )         ( Runtime: -- portb n )
pin-c  ( n "name" -- )         ( Runtime: -- portc n )
pin-d  ( n "name" -- )         ( Runtime: -- portd n )
pin-e  ( n "name" -- )         ( Runtime: -- porte n )
```

For example, you can create a pin designating an error LED and manipulate it using:

```
3 pin-b error-led                   \ Error LED is on port B3
: error error-led bit-set ;         \ Signal error
: no-error error-led bit-clr ;      \ Clear error
```

To ease reading, the words `high`, `low`, `high?`, `low?` and `toggle` are aliases for, respectively, `bit-set`, `bit-low`, `bit-set?`, `bit-clr?` and `bit-toggle`.

You can change the direction of a pin by using `>input` or `>output` after a pin defined with `pin-x`. For example, to set the error led port as an output, use:

```
error-led >output
```

## 5.20 Watchdog timer

The word `clrwdt` is available from Forth to clear the watchdog timer.

## 5.21 Reading from or writing to EEPROM

By using

```
include piceeprom.fs
```

you have access to new words allowing you to access the PIC EEPROM:

```
ee@          ( a -- b )     read the content of a and return it
ee!          ( b a -- )     write b into a
```

Also, in any case, you can store data in EEPROM using those words:

```
eecreate     ( "name" -- )            similar as create but in
                                      EEPROM space
ee,          ( b -- )                 store byte in EEPROM
s"           ( <ccc>" -- eaddr n )    store string in EEPROM
l"           ( <ccc>" -- eaddr n )    strore string + character 13
                                      in EEPROM
```

## 5.22 Reading from or writing to flash memory

Two words allow reading from and writing to the flash memory when the file '`picflash.fs`' is included with

```
include picflash.fs
```

Those words expect manipulate a 14 bits program memory cell whose 13 bits address is in EEADRH:EEADR. The data is read from or stored to EEDATH:EEDATA.

```
flash-read ( -- )
flash-write ( -- )
```

If '`picisr.fs`' has been included before this file, interrupts will be properly disabled around flash writes.

## 5.23 Map and disassembler code

A map can be generated in interactive mode using the `map` word.

## 5.24 Multitasking

Two multitasker have been implemented.

### 5.24.1 Priority-based multitasker

A basic priority-based cooperative multitasker allows you to concurrently run several indenpendant tasks. Each task should execute in a short time and will be called again next time (the entry point does not change). This looks like a state machine.

To use this multitasker, use `include priotasker.fs` in your program.

The following words can be used to define tasks (the entry point for the task is the next defined word):

```
task ( prio "name" -- )
                Define a new task with priority prio. By default, this
                task will be active. You can use the start and
                stop words to control it. Those words can be
                used from an interrupt handler.

task-cond ( prio "name" -- )
                Define a new task with priority prio. By default, this
                task is inactive. You can enable it by using the
                signal word on it. If you use signal N
                times, then the task will be run exactly N
                times. signal can be used from an interrupt handler.

task-idle ( -- )
                Define a new task which will be executed
                inconditionnaly when there is nothing else to do. Such
                a task can not be stopped.

task-set ( bit port prio -- )
                Define a new task with priority prio that will be run
                when bit bit of port port is set.

task-clr ( bit port prio -- )
                Define a new task with priority prio that will be run
                when bit bit of port port is clear.
```

Priority 0 is the greatest one, while priority 255 corresponds to the lowest (idle) priority. You should use priority in the range 0-254 for your own tasks.

The multitasker is run by using the word `multitasker`. This word takes care of scheduling the highest priority tasks first. It also clears the watchdog once per round.

The multitasker looks for all tasks of priority 0 ready to execute. If it find some, it executes them and starts over. If it doesn't, it looks for priority 1 tasks ready to execute. If it find some, it executes them and starts over. If it doesn't, etc. It does this up to priority 255.

Since each word is called each time from the beginning, there is no need to maintain task-specific stacks, as the stack has to be considered empty.

### 5.24.2 Basic cooperative multitasker

The basic cooperative multitasker is much simpler. It allows you to relinguish the CPU whenever you want, provided that you are not in the middle of a call (context-switch only occurs during top-level calls).

To use this multitasker, use `include multitasker.fs` at the top of your program. The following words are defined:

```
task ( -- )
    Create a new task with its own data stack. The task entry point
    will be the next defined word.

yield ( -- )
    Relinguish control so that another task gets a chance to
    execute.

multitasker ( -- )
    Code for the multitasker program. This word never returns.
```

This multitasker makes no use of the return stack at all. However, each task takes four to six program words for initialization and five program words to resume the task, plus three or four program words per yield instruction. Context-switching takes at most 18 instruction cycles (3.6 microseconds max on a 20MHz PIC, 18 microseconds on a 4MHz PIC), and typically 14. Also, the multitasker takes care of clearing the watchdog timer at each round.

Each task needs 3 bytes in RAM to save its context and 8 bytes for its data stack.

## 5.25 Libraries

Some libraries can be used to enhance your application:

- '`libnibble.fs`' nibbles and characters conversion
- '`libcmove.fs`' implementation of ANS Forth `cmove` word

## 5.26 Configuration word

The configuration can be configured with the following words:

```
set-fosc   ( n -- )       Choose oscillator mode (default: fosc-rc)
   fosc-lp   Low power
   fosc-xt   External oscillator
   fosc-hs   High-speed oscillator
   fosc-rc   RC circuit
set-wdte   ( flag -- )    Watchdog timer enable (default: true)
set-/pwrte ( flag -- )    Power-on timer disable (default: true)
set-boden  ( flag -- )    Brown-out detect enable (default: true)
set-lvp    ( flag -- )    Low voltage programming (default: true)
set-cpd    ( flag -- )    EEPROM protection disable (default: true)
set-wrt    ( flag -- )    FLASH protection disable (default: true)
set-debug  ( flag -- )    In-circuit debugger disable (default: true)
set-cp     ( n -- )       Code protection (default: no-cp)
   no-cp     No protection
   full-cp   Full protection
   xxxxx     Anything you want, with the right bits set
             (see datasheet)
```

## 5.27 Caveats and limitations

This compiler release suffers from the following known limitations. Note that most of them (if not all) will disappear in subsequent releases.

- No interactivity There is no link between the compiler and the target.

# 6 Optimizations

The following optimizations are implemented:

## 6.1 Tail recursion

Tail recursion is implemented at `exit` and `;` points.

```
: x y z ;
```

generates the following code for word x:

```
call    y
goto    z
```

The sequence `recurse exit` also benefits from tail recursion.

## 6.2 Redundant pop/push are removed

For example, the (particularily useless)

```
dup dup drop
```

sequence generates

```
movf    0x00,w
decf    0x04,f
movwf   0x00
```

which in fact corresponds to a single `dup`.

Also, the following sequence

```
drop 3
```

generates

```
movlw   0x03
movwf   0x00
```

while

```
drop 0
```

gives

```
clrf    0x00
```

## 6.3 Direct-access and literal variants

Most operations use direct-access and literal variants when possible. The following sequence

```
9 and
```

generates

```
movlw    0x09
andwf    0x00,f
```

Also, combined with the redundant push/pop eliminations, the following code

```
dup 9 and if ...
```

generates

```
movf    0x00,w
andlw   0x09
btfsc   0x03,2
```

## 6.4 Load, store and operations are mixed

The following sequence (with `current` and `next` being variables)

```
current @ 1+ 7 and next !
```

generates

```
movf    0x3B,w
addlw   0x01
andlw   0x07
movwf   0x3C
```

## 6.5 Condition inversions

Short (one instruction) `if` actions are transformed into reversed conditions. For example, the following word:

```
\ This word clears port a0 if port c2 is high, and sets port b1
\ in any case.
: z portc 2 high? if porta 0 low then portb 0 high ;
```

generates the following code:

```
btfsc    0x07,2  ; skip next instruction if port c2 is low
bcf      0x05,0  ; set port a0 low
bsf      0x06,1  ; set port b1 high
return           ; return from word
```

## 6.6 Bank switch optimizations

The compiler tries to remove useless bank manipulations. The following word

```
:: ee@ ( addr -- n ) eeadr ! eepgd bit-set rd bit-set eedata @ ;
```

generates:

```
bsf      0x03,6     ; select bank 2
movwf    0x0d       ; write into eeadr (in bank 2)
bsf      0x03,5     ; select bank 3
bsf      0x0c,7     ; set bit eepgd of eecon1 (in bank 3)
```

```
bsf     0x0c,0     ; set bit rd of eecon1 (in bank 3)
bcf     0x03,5     ; select bank 2
movf    0x0c,w     ; read eedata (in bank 2)
bcf     0x03,6     ; select bank 0
decf    0x04,f     ; decrement stack pointer
movwf   0x00       ; place read value on top of stack
return
```

## 6.7 Operation retarget

If an operation result is stored on the stack then popped into w, the operation is modified to target w directly.

For example, the following word:

```
: timer ( n -- ) invert tmr0 ! ;
```

generates

```
comf    0x00,w
incf    0x04,f
movwf   0x01
return
```

## 6.8 Bit test operations

If a `and` operation before a test can be rewritten using a bit test operation, it will.

For example, the code:

```
checksum @ 1 and if parity-error exit then ...
```

will be compiled as:

```
btfsc   0x33,0
goto    0x037      ; parity-error
...
```

Using an explicit bit-test holds the same result:

```
porta 3 high? if exit then
```

will be compiled as:

```
btfsc   0x05,3
return
```

## 6.9 Useless loads removed when testing

Before a test, if the z status bit already holds the right result, no extra test will be generated.

```
9 and dup if 1+ then
```

will be compiled as:

```
movlw    0x09
andwf    0x00,f
btfss    0x03,2
incf     0x00,f
```

Also, the compiler detects operation which do not modify neither w or the top of stack. For example,

```
dup checksum xor! dcc-high !
```

will be compiled as

```
movf    0x00,w
xorwf   0x6c,f
incf    0x04,f
movwf   0x5b
```

## 6.10 Increment/decrement and skip if zero used when possible

The following word:

```
: action-times ( n -- ) begin action 1- dup while repeat drop ;
```

will be compiled as:

```
call    0x022          ; call action
decfsz  0x00,f
goto    0x027          ; jump to call action above
incf    0x04,f
return
```

## 6.11 Values are not normalized when this is not necessary

The word:

```
:: x ( n -- flag ) 3 < if a then ;
```

generates

```
addlw   0xFD
btfss   0x03,0
call    a
return
```

The `<` test did not cause the value to be normalized to 0 or -1, as it is not needed.

# Appendix A Examples

Some files are included as examples with a Makefile. E.g, to build '`booster.hex`', run `make booster.fs`:

- '`booster.fs`' code for a booster which handles overload and overheat signals This also serves as an example for the priority-based multitasker.
- '`generator.fs`' code for a DCC signal generator based on serial commands (work in progress, not functional yet)
- '`silver.fs`' code that runs on a silver card (a smartcard with a 16f876 and a 24c64 serial eeprom)
- '`taskexample.fs`' example of multitasking code using the basic multitasker
- '`controller.fs`' another multitasking example, used to control multiple peripherals and inputs using a serial link
- '`i2cloader.fs`' a flash and eeprom loader using an I2C bus to reprogram the PIC
- '`spifcard.fs`' production code used in the Ambience European project; includes i2c code, dialog with a bq2010 chip, interface with a smartcard reader using a TDA8004, interrupt code for implementing an in-house watchdog, working around I2C bugs and blinking a led, and analog to digital conversion

# Table of Contents