

# PIC0

A Programming Language for 10F2xx Series PIC Microcontrollers

Version 1.0

Ron Kneusel  
ron@kneusel.org  
27-Dec-2006

# Table of Contents

Overview.....	3
Why?.....	3
Supported Devices.....	3
Getting Started.....	3
Syntax and Semantics.....	8
Processor Type.....	8
Configuration.....	8
Numbers.....	8
Equates.....	8
Function Definition.....	9
Statements.....	9
Loops.....	10
Conditional Statements.....	10
Library Routines.....	11
Inlined Assembly Code.....	11
Examples.....	11

## Overview

PIC0 (pronounced “pico”) is a simple programming language intended to provide an alternative to pure assembly or C for the 8-bit 10F2xx series microcontrollers from Microchip. PIC0 is a compiler to assembly which is then assembled using MPLAB or gpasm. Since the compiler is written in pure Python it will run equally well on any platform. I called it PIC0 for “pico” meaning very small and PIC-zero meaning it is for the simplest of PIC processors.

## Why?

Why not? It was fun to write and I do use it when possible for the small projects I do with the 10F series chips. In real life I'm a developer of medical imaging software, some hardware stuff but mostly analysis algorithms and image processing, and I play around with these little chips for fun. I don't mind using assembly but thought it would be fun to see if I could make something that made good use of the very limited resources these chips have.

## Supported Devices

At present, the compiler works with these chips: 10F200, 10F202, 10F204, 10F206, 10F220, and 10F222. If no processor is declared in the source code it defaults to 10F200.

## Getting Started

Here's a simple PIC0 program to flash three LEDs connected to GPIO.0, GPIO.1, and GPIO.2. The circuit is trivial: connect one LED each to pins 3, 4, and 5 of the PIC, then to ground. Connect +5V to Vdd (pin 2) and ground to Vss (pin 7).

The PIC0 source code is (line numbers added):

```
001 ; Processor
002 p10f206
003
004 ; Config
005 IntRC_OSC
006 WDT_OFF
007 CP_OFF
008 MCLRE_OFF
009
010 ; Equates
011 equ[
012     test    0x12 ; LED pattern
013     delay  0x13 ; outer delay counter
014     delay0 0x14 ; inner delay counter
015 ]
016
017 ;-----
018 ; wait
019 ;
020 [ wait ; ( delay -- )
021     delay!
```

```

022     {
023         0 delay0!
024         {
025             delay0--
026             delay0@ ?0break
027         }
028         delay--
029         delay@ ?0break
030     }
031 ]
032
033 ;-----
034 ; lights
035 ;
036 [ lights ; ( pattern -- )
037     test!
038     GPIO/2 GPIO/1 GPIO/0 ; turn all LEDs off
039     if(test^2) GPIO^2 then
040     if(test^1) GPIO^1 then
041     if(test^0) GPIO^0 then
042 ]
043
044 ;-----
045 ; main
046 ;
047 [ main
048     W->R0 OSCCAL!           ; store oscillator calibration value (in W)
049     OSCCAL/0                ; disable INTOSC/4 on GPIO.2
050     0 GPIO!                 ; clear GPIO
051     0 CMCON0!               ; disable comparator
052     0b00001000 R0->W tris   ; set GPIO directions
053     0b11000000 R0->W option ; set options
054
055     ; Loop forever
056     {
057         0b100 lights 100 wait
058         0b010 lights 100 wait
059         0b001 lights 100 wait
060     }
061 ]

```

this source code should be in the distribution as `flash.pic0`.

A PIC0 program consists of a series of tokens, whitespace and comments. Tokens are non-whitespace characters, ie, printing characters, separated by one or more spaces, newlines, or tabs. Comments begin with a “;” and extend to the end of the current line. There are several sections to a PIC0 program, and they can appear in any order. All variables are global and are declared in the “equ[ ... ]” section using the equate name followed by the memory location assigned to it. See lines 011..015 above. Numbers in PIC0 are assumed decimal unless they are of the form 0xff for C style hexadecimal or 0b1011 for binary. All numbers are considered unsigned and must be in the range [0,255]. PIC0 programs can declare the processor that will be used by placing a P10F2xx token in the source outside of a function or equate declaration. Also, the configuration for the processor, those things given on the `__CONFIG` line in the assembly code, can be given in a like manner using the names, case-sensitive, that the assembler expects. See lines 001-008 above.

Functions are declared using this syntax:

```
[ <name> <body> ]
```

where <name> is the name of the function and <body> are the statements making up the body of the function. There must be, at minimum, a “main” function declared, as in C. This function can be declared anywhere in the source code. The example above defines three functions: main, lights, and wait. So, the smallest possible PIC0 program is: [ main ] which will declare an empty main function and nothing else.

Let's take a closer look at the main function:

```
047 [ main
048   W->R0 OSCCAL!           ; store oscillator calibration value (in W)
049   OSCCAL/0               ; disable INTOSC/4 on GPIO.2
050   0 GPIO!                ; clear GPIO
051   0 CMCON0!              ; disable comparator
052   0b00001000 R0->W tris  ; set GPIO directions
053   0b11000000 R0->W option ; set options
054
055   ; Loop forever
056   {
057     0b100 lights 100 wait
058     0b010 lights 100 wait
059     0b001 lights 100 wait
060   }
061 ]
```

the first two instructions (line 048) take the current contents of the W register, which will be the oscillator calibration value, and put it into the OSCCAL register. The PIC0 compiler is aware of all the standard names for registers, etc. and will even flag when a register is used that isn't available on the given processor. For example, trying to use A/D registers with a 10F204, etc.

PIC0 supports up to two implied arguments to functions. It declares the first two RAM locations as R0 and R1 for register zero and one. These are loaded implicitly by referencing a number or contents of an equate with the compiler handling the assignment. So, the W->R0 instruction will take the contents of the W register and place it into R0. Then, the assignment to OSCCAL is done using “!” as a suffix to indicate how the equate should be used. In this case, “!” means to take the contents of R0 and put it into the OSCCAL register.

Line 049 clears bit zero of the OSCCAL register. Bit reference, set, and clear instructions use suffixes on equates with “.” for bit reference (to R0), “^” for bit set, and “/” for bit clear. For example,

```
myequ.3   put the value of bit 3 of myequ into R0
myequ^6   set bit 6 of myequ
myequ/1   clear bit 1 of myequ
```

Direct assignment to an equate is done by placing the number in R0 and using the “!” suffix. This has a flavor similar to Forth but PIC0 is not stack-based and there can be no space between the equate and the “!” suffix. Therefore, lines 050 and 051 set GPIO and CMCON0 to zero, respectively.

Lines 052 and 053 set up the chip configuration for the TRIS and OPTION registers. They use the R0->W command which is the opposite of W->R0. So, 0b1000 (decimal 8) is placed into W so that it will

be used by the `tris` instruction and `0b11000000` (decimal 192) is placed into the `OPTION` register. A few instructions will be saved by using inlined assembly code here and in the “`W->R0 OSCCAL!`” statement in line 048. See the file `dice.pic0` for an example of this.

Let's look at the `lights` function as it introduces some new syntax:

```
036 [ lights ; ( pattern -- )
037     test!
038     GPIO/2 GPIO/1 GPIO/0 ; turn all LEDs off
039     if(test^2) GPIO^2 then
040     if(test^1) GPIO^1 then
041     if(test^0) GPIO^0 then
042 ]
```

This function expects one argument in `R0`. If we look at `main` again we see that before `lights` is called a number is placed into `R0`. This is the usual way to supply an argument to a function. If a second argument is required they would both be given before the function name. If another number is given after the second the compiler “wraps” around and that number replaces the first one put into `R0`. The compiler is able to see when a function or other action has happened and resets this so that the proper register, either `R0` or `R1`, will be set the next time. If a function leaves a result in `R0`, that result can be preserved by using an underscore (“`_`”) character. So, for example, to add two numbers and then subtract a number from their result: `1 2 + _ 4 -` since the “`_`” causes the 4 to be put into `R1` and not `R0`.

So, the argument to `lights` is stored in `test`. This is strictly unnecessary as nothing in `lights` would cause `R0` to be over written so it would be possible to save a few instructions by replacing “`test`” with “`R0`” in this case. After storing the pattern of LEDs all three LEDs are turned off by clearing the respective bits in `GPIO` (line 038). Then, bits of `test` are checked to see if that LED should then be turned back on. This is the new syntax:

```
if(test^2) GPIO^2 then
```

the first token is a bit-test `if`. There can be no whitespace in this statement because of the parsing rules. The form is:

```
if(<equate><op><bit#>)
```

with every part required. The `<equate>` is a defined `equate` or system `equate` (ie, those defined in the include files used by the assembler). The operation, `<op>` is either a “`^`” for testing if the given bit is set or a “`/`” for testing if a given bit is clear. This mirrors the bit set and bit clear suffixes. Naturally, `<bit#>` is the bit number to test, `0..7`, and must be given in decimal. Only if the `if(...)` is true will the body of the `if` statement be executed (everything up to the “`then`”). In this case, if a particular bit of `test` is set, the corresponding bit of `GPIO` will be set thereby turning that LED on.

The `wait` function illustrates loops:

```
020 [ wait ; ( delay -- )
021     delay!
```

```

022     {
023         0 delay0!
024         {
025             delay0--
026             delay0@ ?0break
027         }
028         delay--
029         delay@ ?0break
030     }
031 ]

```

This function just kills time (a timer could have been used here, of course). There are two loops here denoted by “{“ and “}”. Loops in PIC0 repeat infinitely. You have to break out of them using one of the break keywords. The outer loop counts `delay` down to zero while the inner loop counts from 255 down to 0 each time. Looking at the inner loop:

```

023     0 delay0!
024     {
025         delay0--
026         delay0@ ?0break
027     }

```

we see `delay0` set to 0 and then a loop that breaks when `delay0` is exactly zero. The `?0break` keyword breaks out of the inner-most loop when the value of `R0` is 0. The `?break` instruction would break when the value is 1 (not non-zero). There is also an unconditional `break` and corresponding `continue` instructions: `cont`, `?cont`, `?0cont`. Yes, one could replace `?0break` with `0= if break then` but the former saves many instructions as it doesn't require including the library function `0=`, a subroutine.

To compile the file, thereby creating `flash.asm`, use:

```
python pic0.py flash.pic0
```

assuming Python to be properly installed and in your path. For Windows, you can copy the Python executable to the [C:\WINDOWS](#) folder.

To assemble the output use MPLAB or install `gpasm`, the GNU version, which runs just fine under Windows and Unix:

```
gpasm flash.asm
```

the resulting `flash.hex` file is now ready to download to a 10F206 chip. Connect the LEDs and power and all should be well. The resulting object file uses 73 of the 512 memory locations on the chip.

# Syntax and Semantics

A complete description of the syntax and semantics for PIC0 is given here.

## **Processor Type**

A single token, outside of any function and equate definition, which tells the compiler which processor is the target. Only the first one found is used. If not present, the compiler assumes the target is the 10F200. Possible tokens are (case irrelevant): P10F200, P10F202, P10F204, P10F206, P10F220, and P10F222.

## **Configuration**

Configuration parameters as given in the `__CONFIG` section of an assembly file, with one additional. Must be given as listed, case matters: `IntRC_OSC`, `WDT_OFF`, `CP_OFF`, `MCLRRE_OFF`, plus any others understood by the assembler. Must be given as tokens anywhere in the source but outside of an equate or function definition.

The additional configuration option is `no_sleep` which tells the compiler to not end the main function with a sleep instruction. This could have been used in the example above to save one instruction since the main function enters an infinite loop.

## **Numbers**

All numbers are unsigned bytes, 0..255. If no prefix is given they are assumed to be decimal. A `0x` prefix indicates hexadecimal and a `0b` prefix indicates binary. Numbers are used as arguments to functions and as values assigned to equates. Bit operations and bit-if statements must use equates, not numbers.

## **Equates**

Equates associate a number with an identifier. They are defined in `equ [ . . . ]` sections which can appear at any position in the source code. The body of the equate section is a set of pairs of tokens: `<equate> <value>`. For example,

```
equ[ aaa 0x11   bbb 0b1111   ccc 10 ]
```

defines three equates and associated values.

## Function Definition

Functions are defined as: [ <name> <body> ] with <name> being the name of the function and <body> being the statements that make up the function. All functions are compiled to subroutine calls and may make use of library routines which are inlined whenever possible. A function may call another function if the first function has been called from `main` and the second function does not call another user-defined function nor a library routine that has been implemented as a function. This second condition might be a bit tricky as while some library routines are always implemented as subroutines (eg, `0=`) others are inlined unless referenced too often in which case they are implemented as subroutines as well. All in all, be careful when a function calls a function. This is a consequence of the two-level hardware return stack which is part of all 10F2xx series chips. See “Getting Started” above for examples of function definitions. At a minimum, a `main` function must be defined. The name of a function, or equate for that matter, is restricted to what the assembler will recognize as a valid label.

## Statements

PIC0 is simple enough that the term “statements”, aside from loops and conditional statements, encompasses only function calls, bit reference, set, and clear operations, and storing and fetching from an equate. Two arguments are tracked automatically by the compiler. This means that to call a function that accepts one argument, which will be in `R0`, use:

```
123 one_arg
```

to call a function that accepts two arguments, which will be in `R0` and `R1`, respectively, use:

```
123 234 two_args
```

if three or more arguments are required, use an equate to hold them. If a function returns a value it is placed in `R0`. Library routines do this. If a user-defined function is to return a value it must place it in an equate, which might be `R0`, or `R1`, or both to return two values. Recall that `R0` and `R1` are declared by the compiler to occupy the first two RAM locations of the given processor. Use an underscore, “`_`”, to indicate that whatever is in `R0` should stay there and that the next argument should be put into `R1` (two “`_`” will preserve both `R0` and `R1`). This is useful for expressions: `1 2 + _ 1 -`, etc.

To store the current value of `R0` in an equate place a “`!`” suffix: `123 A!` The compiler is smart enough to optimize this operation to save unnecessary instructions. Similarly, to return the value in an equate use an “`@`” suffix: `A@ myfunc` to place the contents of `A` in `R0` and then call `myfunc`.

Bit operations include returning the current value of a specific bit of an equate (including system equates), setting a specific bit and clearing a specific bit. These are done via suffixes on the equate name:

```
abc.3          put the value of bit 3 of abc into R0
abc^3          set bit 3 of abc
```

abc/3

clear bit 3 of abc

## Loops

Loops in PIC0 are defined as: { <body> } where <body> is the statements that make up the body of the loop. Loops can be nested. All loops are infinite and must be exited by one of the break instructions. The next iteration of a loop can be done by one of the continue instructions.

The break instructions are:

break            unconditional break from the inner-most loop  
?break          break only if the current value of R0 is 1  
?0break        break only if the current value of R0 is 0

with corresponding continue instructions:

cont            jump from here to the head of the inner-most loop  
?cont          jump to head of loop only if R0 is 1  
?0cont        jump to head of loop only if R0 is 0

equivalent syntax (using C syntax) for various loop structures are:

```
for(i=10; i > 0; i-)        ==> 10 i! { i myfunc i-- i@ ?0break }  
  myfunc(i);
```

```
for(i=0; i < 10; i++)      ==> 0 i! {  
  myfunc(i);                    i@ myfunc i++  
                                  i@ 10 = ?break }
```

```
while (i == 2)             ==> { i 2 = ?0break myfunc i! }  
  i = myfunc();
```

## Conditional Statements

The traditional IF .. THEN .. ELSE structure is supported in PIC0 but using the Forth style of:

```
<condition> if <true_part> else <false_part> then
```

where <condition> is any code that leaves a 1 or 0 in R0 (true or false). If the <condition> is true (1) then the <true\_part> of the statement is executed. If not, then the <false\_part> is executed, if given. The if statement ends with the then keyword. An alternate keyword for if is present, 0if. This tests for R0 being 0 instead of 1 and if so, the <true\_part> is executed. If statements may be nested.

# Library Routines

## Inlined Assembly Code

Sometimes it is necessary to include pure assembly code in a function. This is done using the `asm{ . . . }` syntax where everything between the “`asm{`” and “`}`” tokens is passed as is to the output file using “`\`” as a line break. For example, since the microcontroller starts up with the oscillator calibration value in `W` the first instruction ought to put that value in the `OSCCAL` register. The example above used:

```
W->R0 OSCCAL!
```

Which compiles to 3 instructions:

```
movwf R0
movf R0,w
movwf OSCCAL
```

where with inline assembly two of these instructions can be eliminated by using:

```
asm{ movwf OSCCAL }
```

similarly, the instructions in the example above which set the `TRIS` and `OPTION` registers could be done more effectively in assembly:

```
asm{ movlw b'00001000' \ ; set GP3 as input, others output
      tris GPIO \
      movlw b'11000000' \
      option }
```

thereby saving several instructions. See `dice.pic0`. Note that lines in the inlined assembly code are separated by backslash characters, “`\`”.

## Examples

Several example program are included in the distribution:

<code>flash.pic0</code>	flash three LEDs, on after the other
<code>dice.pic0</code>	a binary “dice” with a switch connected to <code>GPIO.3</code> and LEDs on <code>GPIO0,1</code> , and <code>2</code>
<code>stop.pic0</code>	the same circuit as in <code>flash.pic0</code> but using green, yellow, and red to act as a stoplight
<code>timer.pic0</code>	an example of a flashing LED using the timer module of a <code>10F200</code>

These illustrate most of the features of `PIC0`.