LABORATOIRE DE MICRO-INFORMATIQUE EPFL

Programming the Microchip-PIC microcontrolers

J.D. Nicoud, LAMI-EPFL (now DIDEL nicoud@didel.com Tel +41 21 728-6156)

July 1999, 2nd revision Sept 2003 File www.didel.com/doc/PicSoft.pdf

Table of contents

1. Introduction

The Microchip PIC family of microcontrollers is advertised as easy to learn due to its mere 33 instructions (RISC-like). However, one needs to spend some time to fully understand its architecture, specific features, and the rather unusual mechanism of several instructions. Programming in PIC assembly is quite different from the other well known microcontrollers, such as the 8051, the HC11, or the 68XXX. The benefits of using the PIC family have proven to be worthwhile. They are easy to purchase even in small quantities, inexpensive, they come in small devices (as small as 8-pin and as light as 59 mg), have minute power requirements, high execution speeds, code protection, offer lower cost third party development tools, etc.

The Microchip PIC family is quite large, and constantly getting larger. The low end offers some very interesting 8-pin microcontrollers (uC), known as the 12CXXX family. The 12F675 is the first 8-pin reprogrammable circuit, with a lot of advantages. The original low end 18 - 28 pin 16CXX family proved to be the inspiration of electronic project and gadgets builders over the last five years or so, and the in-circuit reprogrammable (EEPROM based) 16C84/16F84 is an old dream come true. We will focus only on the 14-bit instruction compatible family. We will not consider here the 12-bit instruction processors (16C5x and 12C5xx family) and the 16-bit instructions processors (18Fxx family). The objective of this paper is to explain the PIC hardware and software to a reader already familiar (but not expert) with the PIC, and show him all the software specialties of the PIC, and how to efficiently program some real-time applications, where the PIC excels. More and more PIC devices incorporate specific hardware functions, such as A/D converters, pulse-width modulators, LCD drivers, etc. that will help with the design of smaller, simpler, and more cost effective future products.

We will use the CALM (Common Assembly Language for Microprocessors) notations for easier and more readable code. Microchip notations for instructions will also be used in program examples for those familiar with the PIC assembly language. CALM has been designed to provide beginners with a consistent and uniform set of notations, regardless of the target processor. Switching from one processor to another between designs is therefore an easy task. CALM notations have been defined for about 20 processors. A single page reference card for each processor is all that is needed for the commonly used instructions, when the processor architecture has been understood. CALM assemblers, the Smile-NG editor/assembler/downloader (Windows only) and the Picolo editor/assembler/downloader/translator (Linux/Macintosh/Windows) are freeware, downloadable from the DIDEL site; these excellent software were written at the EPFL (Swiss Federal Inst. of Technology, Lausanne).

At first you might object to the fact that CALM has a more complex notation than Microchip and requires more typing for the same instructions. This is true. However, the advantage of self-documented instructions (common to any processor you use), and the rather short time spent typing, compared to the usually lengthy debugging time, makes CALM attractive. CALM users always become CALM lovers.

2. Program memory

One of the peculiarities of the PIC family is the program memory address field. Although the program address field is 13 bits wide (8k max), the jump and call instructions provide only 11 bits of addresses (2k memory) and on lage processors, there are up to 4 program memory pages, and it is not convenient to jump from one page to another. An additional 8-bit page restriction allows efficient computed jumps and table accesses. On the 12-bit processors, there are several more restrictions (see www.didel.com/picg/doc/PicCompati.pdef) and if a large production is not expected, one should consider only the 12F and 16F families. The 18F family is recommended for large applications mostly programmed in C.

3. Registers

PIC architecture is based on a work register W and a set of general purpose PIC registers (25 to 368). W register should not be considered as an accumulator, similar to the HC11 A,B registers for instance. Other specific registers, such as the Program counter, or the Status register, are just register locations. Most of the operations are performed with registers. Since there is only one operand possible in an instruction, ininitializing a register with a constant (literal, immediate value) have to go through the W register first. The general purpose registers are split into several banks in the larger PIC devices. Special attention should be given to addressing those register banks. In choosing one PIC processor over the other, the program memory size as well as the amount of the general purpose registers is always considered together with the number of I/Os, the package size and the price.

4. Flags

Three particular Status register flags (Carry, Zero, and Digit carry) can be read, written, and tested in conditional 'skip' instructions (section 5.7). Special attention should be given to these flags after instructions because some of them set/clear differently as compared to other non-PIC processors. This might allow for some neat programming tricks but also for some hard-to-find bugs.

The Zero flag behaves normally, but the Carry flag has peculiarities (see section 5.3). The Digit carry D relates to the "half-carry" bit found in other processors but there is no decimal adjust instruction to facilitate BCD (see sections 6.3 and 7.2).

There are no explicit instructions for conditional jumping such as "jump if negative, jump if less than", etc. Additions and subtractions are often necessary prior to using a "skip" instruction for the conditional jump (see section 5.6)

The bits in the status register can be set, cleared, and tested like all other I/O port and register bits. CALM PIC assembler supports conditional skip instructions, similar to the conditional jump instructions found in other processors, proposed by Microchip as macros (see section 5.6).

5. Instructions

Most of the PIC instructions operate on one or two operands. All the register involving instructions reserve a bit for the destination of the result. With the Microchip assembler, the default destination (if the bit is not specified) is the general purpose register, not the W register. Microchip writes f,0 and f,1 notations to specify if the destination is W, or the f register. Other assemblers use f,W, f,1, or f alone for the same purpose. This is very confusing, and CALM avoid a lot of confusions with its explicit operands.

Attention should be given to the CALM "move" instruction. It is similar to the Motorola instruction but different from the Intel's "mov". With CALM, the second operand is the implicit destination operand. For example, Sub B,A (subtract A from B), means A - B --> A. Also Comp B,A (not a PIC instruction) means A is compared to B (a subtraction is performed internally). It will be seen later that the PIC does not follow the Pdp11/Motorola habits (sections 5.3 and 6.2).

5.1. Move instructions

It is important to notice that the status flag Z is not modified by all the move instructions. The notation [Z] means that the Z bit is updated by the instructions: Z=1 if the result of the data transferred is zero, Z=0 otherwise.

MOVLW VAL	Move	#Val,W	[none]
MOVWF REG	Move	W,Reg	[none]
MOVF REG,0	Move	Reg,W	[Z]
MOVF REG,1	Test	Reg	[Z]

The last instruction can be written in CALM "Move Reg,Reg". It is indeed a one-operand that does not modify the content of the register Reg, and may be useful to test if the content is zero or not. One should be careful that "Move W,Reg" does not modify Z, but "Move Reg,W" does it.

There are other special "move" instructions which involve OPTION and direction ("TRIS") registers.

TRIS PORTA	Move	W,TrisA	[none]
TRIS PORTB	Move	W,TrisB	[none]
TRIS PORTC	Move	W,TrisC	[none]
OPTION	Move	W,Option	[none]
CLRWDT	ClrWDT		[none]
SLEEP	Sleep		[none]

On the 40-pin processors, the direction of PortD and PortE cannot be set by a TRIS instruction. The direction registers TrisD and TrisE are only accessible in Bank1.

5.2. Logical instructions

All these instructions modify the status flag Z: Z = 1 if the result of the operation is zero.

ANDLW VAL	And	#Val,W	[Z]
ANDWF REG,0	And	Reg,W	[Z]
ANDWF REG,1	And	W,Reg	[Z]
IORLW VAL	Or	#Val,W	[Z]
IORWF REG,0	Or	Reg,W	[Z]
IORWF REG,1	Or	W,Reg	[Z]
XORLW VAL	Xor	#Val,W	[Z]
XORWF REG,0	Xor	Reg,W	[Z]
XORWF REG,1	Xor	W,Reg	[Z]

5.3. Arithmetic instructions

The add instruction is straightforward; the three status flags are modified: [C,D,Z].

ADDLW VAL	Add	#Val,W	[C,D,Z]	Val + (W) --> W
ADDWF REG,0	Add	Reg,W	[C,D,Z]	(Reg) + (W) --> W
ADDWF REG,1	Add	W,Reg	[C,D,Z]	(W) + (Reg) --> Reg

For subtraction, one needs to understand how the operation is performed internally. Contrarily to other processors, the second operand is subtracted (by adding its 2's complement) from the first operand (immediate value or register content). This is better expressed by 3-operand instructions: diminuand, diminutor, and result.

SUBLW VAL	Sub	W,#Val,W	[C,D,Z]	Val - (W) --> W
SUBWF REG,0	Sub	W,Reg,W	[C,D,Z]	(Reg) - (W) --> W
SUBWF REG,1	Sub	W,Reg	[C,D,Z]	(Reg) - (W) --> Reg (equiv. to Sub W,Reg,Reg)

The Sub instruction must be used to compare the contents of W or a variable, as shown in section 6.2. If the Carry is set, this indicates that the result is positive or that W is lower than the given immediate value or register content. If the Carry is clear, W is higher. The Z flag is set in the case of equality. For equality comparizons, a XOR is quite adequate.

5.4. Increment, Decrement, Complement and Clear

Increment and decrement are not possible on the W register. One can, however, use Add #1,W and Add #-1,W, which modify the Carry, Z, and D flags, while Inc and Dec on a register modify only Z. Sub W,#1,W performs an unusual operation (it adds to value 1 the complement of W content), but Sub W,Reg with a #1 in W can be used to decrement a register (with only the advantage of modifying Carry).

INCF REG,1	Inc	Reg	[Z]	(Reg)+1 --> Reg
INCF REG,0	Inc	Reg,W	[Z]	(Reg)+1 --> W (Reg) not modified
DECF REG,1	Dec	Reg	[Z]	(Reg)-1 --> Reg
DECF REG,0	Dec	Reg,W	[Z]	(Reg)-1 --> W (Reg) not modified
COMF REG,1	Not	Reg	[Z]	/(Reg) --> Reg
COMF REG,0	Not	Reg,W	[Z]	/(Reg) --> W (Reg) not modified
CLRF REG	Clr	Reg	[Z=1]	
CLRW	Clr	W	[Z=1]	

The complement instruction is a logical Not (inversion of all bits) and not a negate (2's complement) instruction. There are two solutions for implementing the negate (2-s complement) instruction found in other processors:

SUBLW 0	Sub	W,#0,W	[C,D,Z]	-(W) --> W
COMF REG,1	Not	Reg		
INCF REG,1	Inc	Reg	[Z]	-(Reg) --> Reg

Not W is implemented with the Xor #-1,W instruction.

5.5. Rotate and Swap

Only two shift instructions are provided, and they do a rotate through carry. This is convenient to examine one bit at a time, but in many cases the carry value must be prepared before the shift. Swapping the two nibbles of an 8-bit variable is possible.

RRF REG,1	RRC	Reg	[C]	
RRF REG,0	RRC	Reg,W	[C]	(Reg) not modified
RLF REG,1	RLC	Reg	[C]	
RLF REG,0	RLC	Reg,W	[C]	(Reg) not modified
SWAPF REG,1	Swap	Reg		
SWAPF REG,0	Swap	Reg,W		(Reg) not modified

It is usually necessary to prepare the carry bit before any rotate. For instance, an 16-bit divide by 2 is written

BCF STATUS,0	ClrC		
RRF High,1	RRC	High	
RRF Low,1	RRC	Low	[C]

5.6. Bit instructions

The PIC is quite powerful to set, clear, and test bits on ports, variables, the processor status register, and even the program counter. The bit number (7..0) specifies the modified bit.

BCF REG,bNumber	Clr	Reg:#bNumber	[none]
BSF REG,bNumber	Set	Reg:#bNumber	[none]
BTFSC REG,bNumber	TestSkip,BC	Reg:#bNumber	[none]
BTFFS REG,bNumber	TestSkip,BS	Reg:#bNumber	[none]

The number sign is consistent with CALM syntax, since it is an immediate value.

Inverting a single bit is possible with the XOR instruction, in order to replace the missing Not Reg:#bNumber instruction.

MOVLW 2**bNumber	Move	#2**bNumber,W
XORLW REG	Xor	W,Reg

Logical instructions can be used to to modify a single bit of W (they can also modify several bits at the same time):

ANDLW -1-(2**BNUMBER)	And	#.NOT.(2**bNumber),W	Clr
IORWF 2**BNUMBER	Or	#2**bNumber,W	Set
XORLW 2**BNUMBER	Xor	#2**bNumber,W	Not

CALM assemblers accept the instructions found in other processors for handling the status flags:

BSF STATUS,0	SetC	[C=1]	Set carry
BCF STATUS,0	ClrC	[C=0]	Clr carry
BSF STATUS,1	SetD	[D=1]	Set D, decimal carry flag (half carry)
BCF STATUS,1	ClrD	[D=0]	Clr D
BSF STATUS,2	SetZ	[Z=1]	Set Zero flag

```
BCF STATUS,2          ClrZ      [Z=0]    Clr Zero flag
BTFSC STATUS,0        Skip,CC     Skip if Carry Clear (result of the
                                    previous ADD,SUB,RLC,RRC)
BTFSS STATUS,0        Skip,CS     Skip if Carry Set
BTFSC STATUS,1        Skip,DC     Skip if Digit carry Clear
BTFSS STATUS,1        Skip,DS     Skip if Digit carry Set
BTFSS STATUS,2        Skip,EQ     Skip if Equal (result of the previous ADD, SUB, INC, ...
                                    operation that modified Z)
                      Skip,ZS     Skip if Zero bit set (same as above)
BTFSC STATUS,2        Skip,NE     Skip if Non Equal
                      Skip,ZC     Skip if Zero bit clear (same as above)
```

Single instruction macros can be used when I/O bits are manipulated, in order to increase the portability of programs. For instance, if an output bit is a serial clock CK, it will be referred in the program by the macros ChOn and CkOf; a chnge of hardware will imply only the modification of the macro:

```
CKON   MACRO              .Macro CkOn
BSF   PORTA,bCK              Set       PortA:#bCk
ENDM                      .EndMacro

CKOFF   MACRO             .Macro CkOff
BCF   PORTA,bCK              Clr       PortA:#bCk
ENDM                      .EndMacro
```

See examples in section 7 for other macros.

5.7. Skip and jump instructions

Conditional jumps usually available with other processors do not exist. Conditional skips are frequently more efficient.

```
INCFSZ   REG,0        IncSkip,EQ   Reg       [none]
                         Increment Reg and Skip if result is Equal to zero
INCFSZ   REG,1        IncSkip,EQ   Reg,W     [none]   (Reg) not modified
                         Copy Reg in W, then increment and Skip if result is Equal to zero
DECFSZ   REG,0        DecSkip,EQ   Reg       [none]
DECFSZ   REG,1        DecSkip,EQ   Reg,W     [none]   (Reg) not modified
GOTO   ADDR           Jump   AddressLabel
```

One can notice that IncSkip,NE etc. are missing. They can be replaced by two instructions:

```
INC   REG             Inc       Reg
BTFSC   STATUS,2      Skip,NE
```

A typical application example is when a counter has to be reloaded at its initial value when it reaches zero. Two instructions are required, it is not possible to skip over both of them. It is hence faster to prepare the value before decrementing, and skip over the reinitialization of the counter if required.

```
                              3 or 4 microsecond duration
    DECFSZ    CNT                 DecSkip,EQ    Cnt
    GOTO       NEXT               Jump        Next
    MOVLW      INICNT             Move        #IniCnt,W
    MOVWF      CNT                Move        W,Cnt
  NEXT                          Next:

                              Always 4 microsecond duration
    MOVLW      INICNT             Move        #IniCnt,W ; in case we need it
    DECF       CNT                Dec         Cnt
    BTFSC      STATUS,2           Skip,NE
    MOVWF      CNT                Move        W,Cnt
```

Programming 16-bit counters is trivial. One can skip the instruction which increment the high bit counter as long the low bit counter does not overflow (that is reaches again zero). An equivalent option is to always increment the high bit counter, but decrement before to compensate when the low bit counter is different from zero. It may looks stupid, but is more efficient for 24-, 32-bit counters.

```
  INCF       CNTLOW              Inc         CntLow    ; least significant byte, bits 7..0
  BTFSC      STATUS,2            Skip,NE
  INCF       CNTHIGH             Inc         CntHigh   ; bit 15..8
or                            or
  INCFSZ     CNTLOW              IncSkip,EQ   CntLow
  DECF       CNTHIGH             Dec         CntHigh
  INCF       CNTHIGH             Inc         CntHigh
```

Decrementing a 16-bit counter need more attention

```
MOVF    CNTLOW,1        Test       CntLow
BTFSC   STATUS,2        Skip,NE
DECF    CNTHIGH         Dec        CntHigh
DECF    CNTLOW          Dec        CntLow
```

5.8. Call and return instructions

There are 8 hardware stack locations for the Program counter and the stack pointer is not accessible. After 8 imbricated calls, the wront return address is taken. The Call and Return instructions of course do not modify the status flags.

```
CALL  LABEL    Call  Label        (call routine)
RET            Ret                (return from subroutine)
RETFIE         RetI               (return from interrupt)
RETLW  K       RetMove #Val,W     (load W and return)
```

\the RetI (return from interrupt) set the GIE (General Interrupt enable) bit in addition to reloading the return address from the stack.

The last instruction, RetMove #Val,W returns the given value in W and is quite interesting; its use will be explained in section 6.5.

6. Specific approach to PIC programming

6.1. Logical AND between bits

It is rather simple with the PIC to do logic operations between any two bits of registers or ports. For the logical AND, one writes

```
BTFSC   REG1,BIT1       TestSkip,BC   Reg1:#Bit1
BTFSS   REG2,BIT2       TestSkip,BS   Reg2:#Bit2
GOTO    DONAND          Jump      DoNand    ; if Bit1 or Bit2 is 0
                        ; Continue here if Bit1 .AND. Bit2 is 1
```

The following set of instructions may be useful, and can be inserted in a library of Macros. The sign $, respectively APC in CALM, is the assembler PC; using this notation avoids the need to declare labels, but is not encouraged.

```
                     ; SkipAND   A,B    Skip next instruction if A.AND.B = 1
BTFSS   REG1,BIT1        TestSkip,BS   Reg1:#Bit1
BTFSC   REG2,BIT2        TestSkip,BC   Reg2:#Bit2
                         (Jump      Nand)

                     ; SkipOR   A,B    Skip next instruction if A.OR.B = 1
BTFSC   REG1,BIT1        TestSkip,BC   Reg1:#Bit1
GOTO    $+2              Jump      APC+2
BTFSS   REG2,BIT2        TestSkip,BS   Reg2:#Bit2
                         (Jump      Nor)

                     ; SkipXOR   A,B    Skip next instruction if A.XOR.B = 1
BTFSC   REG1,BIT1        TestSkip,BC   Reg1:#Bit1
BTFSC   REG2,BIT2        TestSkip,BC   Reg2:#Bit2
GOTO    $+2              Jump      APC+2
GOTO    $+4              Jump      APC+4
BTFSS   REG1,BIT1        TestSkip,BS   Reg1:#Bit1
BTFSS   REG2,BIT2        TestSkip,BS   Reg2:#Bit2
                         (Jump      XNor)
```

6.2. Writing to a port

A I/O port is frequently assigned to several tasks inside the program. One given task that would just write on the port could change bits under the control of another task. Set and Clr bits solve this problem, but is iis not efficient if several bits have to be modified. The best is to read the port, modify only the concerned bits, and write the result on the port.

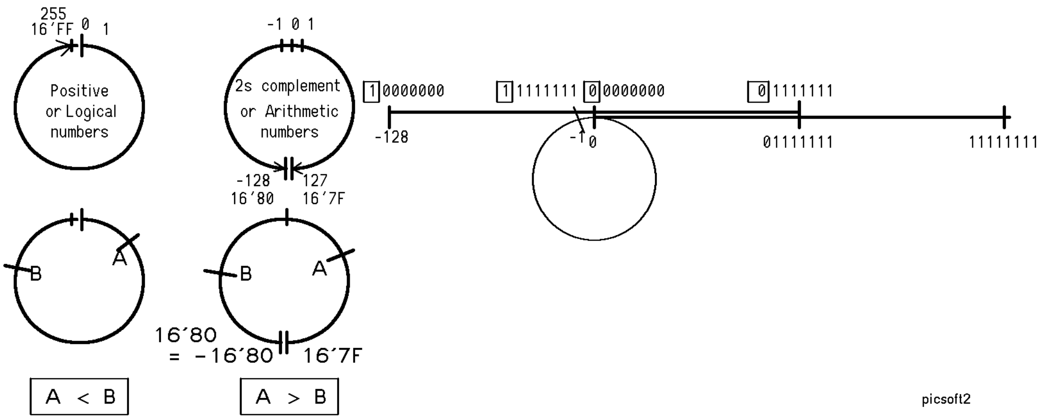
```
Mask      = 2'01110000     ; Bits that may be changed
; New     ; Value to be copied on the port, on bits 6,5,4
          ; Bits 7, 3,2,1,0 are all clear
Move      Port,W
And       Mask.XOR.16'FF     ; Clear the bits 6,5,4
Or        New,Port           ; Update bits 6,5,4
Move      W,Port
```

An interrupt must not modify the port during these instructions.

6.3. Positive and negative numbers

Microcontroller are mostly handling 8 bit positive numbers, value 0 to 255. Overflow in an addition will set the carry. Underflow in a subtraction is not allowed (and will clear the carry in the special case of the PIC).

Negative numbers can be represented from several ways. The sign can be placed in another status word. Usually, the 2-s complement form is used, bit 7 is the sign bit, and numbers range between -128 and +127. When comparing numbers, it is important to know their reprezentation.



6.4. Comparing variables

The compare instruction does not exist on the PIC. For equality, one can use the XOR instruction. When the two operands are equal, the result is zero, and the Zero bit is thus set.

For the usual LO (lower), LS (lower same), HI (higher), HS (Higher same) compare between positive integers, one must subtract, and see the result, taking care of the inverted Carry value (section 5.3). If W's contents are lower than or equal to the compared value, Carry is set. Signed numbers are not considered here, see www.didel.com/picg/doc/Arith.pdef).

Hence, one can define the following instruction groups. They modify W, which is usually not expected from a Compare instruction. As usual with the "source-destination notations" (Motorola), Comp A,B means B is compared to A. Result is HI (higher) if B > A.

```
                                ; Skip_HI (CC) Comp #Val,W and skip if (W) higher than Val (modify W)
SUBLW   VAL                         Sub     W,#Val,W
BTFSC   STATUS,0                    Skip,CC
                                    (jump if lower or same)

                                ; Skip_HS (CC or EQ) Comp #Val,W and skip if higher or same
SUBLW   VAL                         Sub     W,#Val,W
BTFSS   STATUS,0                    Skip,CS
GOTO    $+3                         Jump    APC+3
BTFSS   STATUS,2                    Skip,EQ
                                    (jump if lower)
                                ; Skip_LO (CS and NE) Comp #Val,W and skip if lower
SUBLW   VAL                         Sub     W,#Val,W
BTFSS   STATUS,2                    Skip,EQ
BTFSS   STATUS,0                    Skip,CS
                                    (jump if higher or same)
                                ; Skip_LS (CS) Comp #Val,W and skip if lower or same
SUBLW   VAL                         Sub     W,#Val,W
BTFSS   STATUS,0                    Skip,CS
                                    (jump if higher)
```

It should be noticed that the immediate value can be replaced with a variable.

For checking if W is between two limits Low and High, the usual algorithm works:

If (W) < Low, set Carry
If (W) > High, set Carry
Otherwise clear Carry

This can be implemented with 6 instructions.

One can save several instructions and more cycles by writing the next instructions. The condition is, however, that "Low" is greater than zero (1 or more), and "High" is lower than 16'FF (254 and less). Of course, "Low" is less than or equal to "High".

```
MOVWF   TEMP        Move   W,Temp        ; temporary register
SUBLW   LOW-1       Sub    W,#Low-1,W    ; (Low-1)-(W), CS if positive, Low >= (W)
BTFSC   STATUS,0    Skip,CC
GOTO    $+3         Jump   APC+3         ; exit with Carry Set
MOVLW   HIGH+1      Move   #High+1,W
SUBWF   TEMP        Sub    W,Temp        ; (Temp)-High, CS if positive, (Temp) >= High
```

6.5. Absolute value and saturation

When comparing two variables, the absolute wvalue of their difference is easily calulated, but the difference must be lower than 128 = 16'80.

```
|VAR1-VAR2| --> W          |Var1-Var2| --> W
MOVF    VAR1,0             Move       Var1,W
SUBLW   VAR2               Sub        W,Var2,W
MOVWF   TEMP               Move       W,Temp
BTFSC   TEMP,7             TestSkip,BC Temp:#7   ; sign bit
SUBLW   0                  Sub        W,#0,W
```

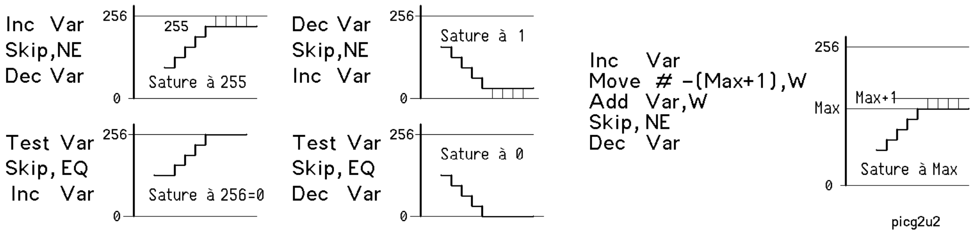
Frequently, a value must stay between limits. For positive numbers, negative saturation at value Min included (any value of the variable lower than Min will be replaced by the value Min) is written:

```
MOVLW   MIN                Move       #Min,W
SUBWF   VAR,1              Sub        W,Var
BTFSS   STATUS,0           Skip,CS
CLR     VAR                Clr        Var
ADDWF   VAR,1              Add        W,Var
```

Positive saturation at value Max included (any value of the variable higher than Max will be replaced by the value Max) is written:

```
MOVLW   MAX                Move       #Max,W
SUBWF   VAR,1              Sub        W,Var
BTFSC   STATUS,0           Skip,CC
CLR     VAR                Clr        Var
ADDWF   VAR,1              Add        W,Var
```

Counters frequently have to saturate as shown in next figure.



6.6. Multiprecision and BCD operations

Multiprecision and BCD is not as easy to handle as with other 8- or 16-bit processors. Microchip documents a set of Utility Math Routines. Multiprecision counters will be given in section 7.2.

16-bit registers are sometimes required because 8-bit operations overflow, as shown in section 6.4. Since the AddC and SubC instructions available in other processors are not implemented, these instructions have just to be emulated.

```
                           ;Add Nb1.16,Nb2.16  (Nb1+Nb2 --> Nb2)
MOVF    NB1LOW,0           Move       Nb1Low,W
ADDWF   NB2LOW,1           Add        W,Nb2Low
MOVF    NB1HIGH,0          Move       Nb1High,W; Must be < 16'FF
BTFSC   STATUS,0           Skip,CC
ADDLW   1                  Add        #1,W
ADDWF   NB2HIGH,1          Add        W,Nb2High
```

6.7. Indirect access

The file select register FSR (at address 4) is a pointer to the register set of the PIC. It can be written and read from address 4. When accessed at address 0, the processor executes an indirect transfer with the pointed-to register. The curly braces are used with CALM to show indirect access.

For instance, if FSR content is 16'12, Move FSR,W will be executed as a transfer of location 4 content to W, and W will get a 12. Move {FSR},W will be translated by the assembler as a move from location 0 to W, but that location does not exist: the processor takes the data inside register 4 as an address, and in our example it will get from location 12 a content to be put in W.

```
MOVF 0,0   Move {FSR},W     [z]
MOVF 0,1   Move W,{FSR}     [none]
```

For instance, if the averave of incoming numbers are to be calculated ovet the last 4 numbers (sliding average), the simplest is to reserve a memory block from address 30 or 40, and increment the pointer circularly by never allowing bit 2 to be set. Old data is taken out of the table and new data introduced. A 16-bit register is required to accumulate the 4 consecutive values (that may be divided by 4 when requred, see section 5.5)

```
MOVWF    0,0                    Move      {FSR},W
SUBWF    W,PpmLow               Sub       W,PpmLow
BTFSS    STATUS,0               Skip,CS
DEC      PpmHigh                Dec       PpmHigh
MOVWF    PpmValue,0             Move      PpmValue,W
MOVWF    0                      Move      W,{FSR}
ADDWF    PpmLow,1               Add       W,PpmLow
BTFSC    STATUS,0               Skip,CC
INCF     PpmHigh,1              Inc       PpmHigh
INCF     FSR,1                  Inc       FSR
CLRF     FSR     ; Clear bit    Clr       FSR:#2    ; Clear bit 2
         2
```

6.8. Tables

There is no addressing mode to access data in program memory, except on the 16F87x family, where the EeAdrH/EeAdr 14-bit register allows to point anywhere in memory. The "RetMove" instruction is a nice trick to access data: that special subroutine return instruction brings back into W the 8-bit value put in the instruction. The routine which returns the n'th value of a table is simply

```
ADDWF  PCL,W     Table:  Add     W,PCL
RETLW  VAL0              RetMove #Val0,W
RETLW  VAL1              RetMove #Val1,W
         ...
```

A macro makes these tables easier to write and read. With the CALM assembler, the macro parameters are not listed with the macro name, they are labeled %1, %2 within the instructions of the macro.

```
DD MACRO VAL      .Macro   DD
    RETLW  VAL             RetMove #%1,W
    ENDM          .EndMacro

TABLE             Table:
  ADDWF 2         Add      W,PCL
  DD   VAL0       DD       Val0
  DD   VAL0       DD       Val1
...               ....
```

Tables are frequently used for replacing function calculation or compensating for nonlinearity. They cannot cross a 256-instruction boundary. It is safe to add a conditional test at the end of a table so the assembler will signal the error:

```
TABLE             Table:
...               ....
???               .If      APC .GT. 16'100    ; Over first page?
! page overflow ! ! page overflow !
???               .Endif
```

The last element of a table can include instructions. E.g., if a table has to be scanned circularly, as it is the case for a stepping motor, the last state will be preceeded by a Clr Pointer instruction (see Section 10.1).

6.9. Computed "goto"

Since the PC (program counter) can be accessed as a register, computed jumps are easy within the current page: the Move W,PCL will continue execution at the address prepared in W. But this is an 8-bit instruction, the PC will stay in the same page of 256 positions! Move W,PC could be written Jump {W}.AND.16'FF+PCLATH*16'100.

A "Jump table" is also easy to implement. For instance, if it is required to jump to Do0, Do1, Do2 according to a variable "Select" taking the values 0, 1, or 2, the table is made of the corresponding jumps.

```
MOVF     SELECT                 Move      Select,W
ADDWF    PCL                    Add       W,PCL
GOTO     DO0                    Jump      Do0
GOTO     DO1                    Jump      Do1
GOTO     DO2                    Jump      Do2
```

It is safe to verify that the table is in the same page, and the PcLath register (section xx) must be prepared it the able is not in the page defined by this PcLath register.

The instruction Add W,PCL can be used to execute a stream of instructions of variable length. For instance, if a pulse of 0 to 5 microseconds (4 MHz processor) must be sent to some pin, loading W with a value between 5 and 0 and executing the following instructions solve the problem.

```
ADDWF   PCL        Add   W,PCL
BSF     PORT,PIN   Set   Port:#Pin
BSF     PORT,PIN   Set   Port:#Pin
BSF     PORT,PIN   Set   Port:#Pin
BSF     PORT,PIN   Set   Port:#Pin
BSF     PORT,PIN   Set   Port:#Pin
BCF     PORT,PIN   Clr   Port:#Pin
```

7. Pages and Banks

7.1. 8-bit pages

There are two page limitation on the PICs. When operations are performed on the PCL register, only the 8 low bits of the PCL are modified, and the 8 upper bits are taken from the PcLatH register. When tables are not all in page zero, it is important to load the PcLath register with the current page value:

```
XXX:  MovLW  $/256      XXX:  Move   #APC/256,W
      MovF   PCLATH,1         Move   W,PcLatH
```

At the end of the table, one check that the page number is still the same

```
.If     APC/256 .NE. XXX/256
! Page error with table xxx
```

7.2. Program pages

Jump and Call instructions on the 14-bit instruction processors have 11 bits for the address. 2k bytes of memory can hece be addressed, up to address 16'7FF. Several PICs have more memory. In this case, one should carefully plan what will be in low memory and what in higer pages. The PcLatH register has to be prepared when jumping from on4e block to another. With Calm, the addresses have to be masked. See www.didel.com/picg/doc/DocPage.pdf (in French) for more details.

7.3. Variable banks

PIC instructions have a 7-bit field for variables. The first 16'20 are used by I/O and control registers. Two bits in the Status register (RP1 RP0) select one of 4 possible register banks. Sometimes a register overlap the 4 banks, usually to access a register in e.g. bank1, one need to ser bit RP0 in Status register, and come back to bank 0 after handling all the registers in bank 1.

8. Simple program examples

Every program can be implemented in several ways. Constraints on execution time, register count, and program size always lead to many variants of the same program module. The objective here is to give the best possible understanding of that flexibility, in order to allow the user to get the best possible performance from their programs.

8.1. Wait loop and counters

Delays are easy to implement with a down-counter. The DecSkip,EQ (DECFSZ, decrement and skip if result is equal to zero) is convenient for this.

The program must start with assembler definitions. Within the PIC family, there are several processors with about the same set of instructions. In some cases, instructions for the PIC programmer have to be inserted. Variables are declared on the Microchip assembler with their absolute position. On CALM, the beginning address of the variables is specified by a .Loc, and then they occupy consecutive locations. It is hence easy to add/remove a variable. As a surprising feature, varaibles must be declared as 16 bits (.16 1 reserve one 16-bit word in memory). The reason is the PIC instructions are 12, 14 or 16 bits, and the universal CALM assembler has to be set in the 16-bit mode.

Program PicTest Oscillate all port bits

```
    LIST    P=16F84
CNT1        EQU 0xC
CNT2        EQU 0xD
    ORG     0
DEB
    MOVLW   0
    TRIS    5
    TRIS    6
    MOVLW   B'10101010'
LOOP
    XORLW   B'11111111'
    MOVWF   5
    MOVWF   6
A
    DECFSZ  CNT1
    GOTO    A
    DECFSZ  CNT2
    GOTO    A
    GOTO    LOOP
    END
```

```
.Proc 16F84
.loc  16'0C      ; DebVar
Cnt1:            .16      1
Cnt2:            .16      1
.Loc  0
Deb:
      Move       #0,W     ; Outputs
      Move       W,TrisA
      Move       W,TrisB
      Move       #2'10101010,W        ; initial value
Loop:
      Xor        #2'11111111,W
      Move       W,PortA
      Move       W,PortB
  A$:            DecSkip,EQ    Cnt1 ; Delay (0.2 s)
      Jump       A$
      DecSkip,EQ     Cnt2
      Jump       A$
      Jump       Loop
.End
```

Initializing Cnt1 and Cnt2 to zero has not been done, since in this case a first waiting loop of a different duration has no importance.

8.2. BCD counter

There are several ways to make a simple decimal counter. One must decide if two digits are packed into a single byte, as below. One digit per byte makes it more straightforward to convert to Ascii or 7-segments, but it is a matter of few instructions and microseconds, and packing saves registers.

The IncBCD routine increments register CntBCD and returns with carry set when the 99 to 00 overflow occurs. The incremented digits are compared to 9 and if the result is larger, a 6 is added in order to bring the digit to zero and add a carry to the higher digit.

With a second program, testing for non-decimal value is done in the same manner as when a hardware asynchronous counter is wired from a binary counter, i.e. an AND gate resets the counter when it reaches state 2'1010. The usage of two consecutive skip instructions has been explained in section 6.1.

```
INCBCD
    INCF    CNTBCD
    MOVF    CNTBCD,W
    ANDLW   B'00001111'
    SUBLW   9
    BTFSC   STATUS,0
    RETURN
    MOVLW   6
    ADDWF   CNTBCD
    MOVF    CNTBCD,W
    ANDLW   B'11110000'
    SUBLW   0x90
    BTFSC   STATUS,0
    RETURN
    MOVLW   0x60
    ADDWF   CNTBCD
    RETURN
```

```
IncBCD:
      Inc        CntBCD
      Move       CntBCD,W
      And        #2'00001111,W     ; Keep LSD (least significant digit)
      Sub        W,#9,W          ; If W > 9, (9-(W) negative) add 6 to CntBCD
      Skip,CC                    ; ! inverted carry
      Ret                        ; no correction
      Move       #6,W
      Add        W,CntBCD
      Move       CntBCD,W
      And        #2'11110000,W     ; Look at MSD
      Sub        #16'90,W
      Skip,CC
      Ret                        ; No correction
      Move       #16'60,W
      Add        W,CntBCD       ; Carry Set if overflow
      Ret
```

```
INCBCD
    INCF    CNTBCD
    BTFSC   CNTBCD,1
    BTFSS   CNTBCD,3
    RETURN
    MOVLW   6
    ADDWF   CNTBCD
    BTFSC   CNTBCD,4+1
    BTFSS   CNTBCD,4+3
    RETURN
    MOVLW   (2**4)*6
    ADDWF   CNTBCD
    RETURN
```

```
IncBCD:
      Inc            CntBCD
      TestSkip,BC    CntBCD:#1
      TestSkip,BS    CntBCD:#3
      Ret
      Move           #6,W
      Add            W,CntBCD
      TestSkip,BC    CntBCD:#4+1
      TestSkip,BS    CntBCD:#4+3
      Ret
      Move           #(2**4)*6,W  ; Another way of handling tens
      Add            W,CntBCD
      Ret
```

If one have to reduce program execution time and enough program memory is available, as is frequently the case, one can just put the "IncBCD" function in a table. Carry and Zero bits are updated correctly. A macro is useful to write the data for the table in a more compact way.

```
    CALL    INCBCD               Call     IncBCD
    ...                          ...
INCBCD                      IncBCD:                          ; Increment CntBCD
    MOVF    CNTBCD,W             Move     CntBCD,W
    CALL    TAINCBCD             Call     TaIncBCD
    ADDLW   1                    Add      #1,W           ; That´s done !
    ...                          Move     W,CntBCD
TAINCBCD                         ...
    ADDWF   PCL,W           ; The table uses 160 program words (min 9x16+10)
                            ; all in the same block of 256 locations
                            ; Macro "DD" is 16 lines long
                            TaIncBCD:
                                 Add      W,PCL
                                 D8       0,1,2,3,4,5,6,7
                                 D8       8,16´0F, 0,0,0,0,0,0    ; 0 to 9, then invalid
                                 D8       16´10,16´11, ...
                                 D8       16´18,16´1F, 0,0,0,0,0,0    ; 10 to 19
                                 ....
                                 D8       16´90,16´91, ...
                                 D8       16´98,16´FF, 0,0,0,0,0,0    ; 90 to 99
                                 .If      (APC.AND.255) .GT. (IncBCD.AND.255)
                                 .Error   Block boundary overflow
                                 .Endif
```

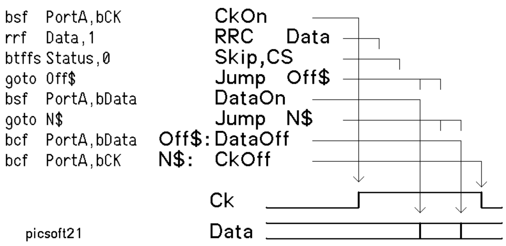
9. Serial transfers

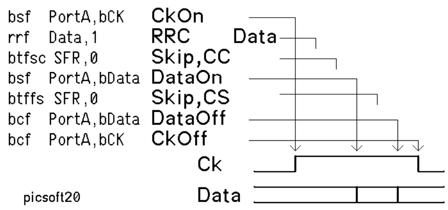
Serial transfers are simple and efficient when the microcontroller is a master (generates the clock) and the slave is a special circuit that can follow the rate imposed by the controller. Errors frequently occur due to a simple naming problem. Inputs and outputs are relative to the considered device. The serial out pin of the microcontroller is sent to the "serial in" input of the device. MOSI (Master out, slave in) is a good name to give the line; "In" or "Out" will eventually generate errors.

9.1. Serial out

Shifting data out or in on a serial port is a typical activity for a microcontroller. The data to be shifted out is prepared in the "Data" variable. The RRC instruction is optimal for taking the decision on all bits in an 8-time repeated loop.

The next figure shows how two branches are taken according to the Carry value. The figure at right is more efficient, but suppose that only one instruction is executed (setting or clearing the data port). It is also to be noticed that the falling edge of the clock is active for transfering the data inside the distant receiver.





The clock polarity and position depends on the application. The above example is intended to show the methodology to be applied, and is not a general solution that can be copied and reused. Rotate direction also depends on the application.

The Write loop can also be written in two different ways. A counter by 8 is the most common.

Routine **Write** **Transfer 8 bit serially**

in: DataOut, transfered LSB first
mod: DataOut, C1

```
WRITE                       Write:
    MOVLW   8                   Move     #8,W
    MOVWF   C1                  Move     W,C1
L                           L$:
    RRF     DATAOUT             RRC      DataOut
    BTFSC   3,0                 Skip,CC
    DATAON                      DataOn   (macro   Set PortA:#bData)
    BTFSS   3,0                 Skip,CS
    DATAOFF                     DataOff  (macro   Clr PortA:#bData)
    CKON                        CkOn     (macro   Set PortA:#bCk)
    CKOFF                       CkOff    (macro   Clr PortA:#bCk)
    DECFSZ  C1                  DecSkip,EQ  C1
    GOTO    L                   Jump     L$
    RETURN                      Ret
```

Setting the carry before the first shift and waiting for the register to be empty is just as fast, and saves a variable.

Routine **Write** **Transfer 8 bit serially at 100 kb/s**

in: DataOut, transfered LSB first
mod: DataOut

```
WRITE                            Write:
    BSF     3,0                      SetC
L                                L$:
    RRF     DATAOUT                  RRC         DataOut
    MOVF    DATAOUT,W                Move        DataOut,W
    BTFSC  STATUS,2                  Skip,NE
    RETURN                           Ret
    BTFSC   3,0                      Skip,CC
    DATAON                           DataOn
    BTFSS   3,0                      Skip,CS
    DATAOFF                          DataOff
    CKON                             CkOn
    CKOFF                            CkOff
    BCF     3,0                      ClrC
    GOTO    L                        Jump        L$
```

If the clock pulse must be longer, it may influence the selection of one of these two schemes.

9.2. Serial in

Shifting data in uses a quite similar mechanism. According to the data bit read, a "0" or "1" is shifted into the data register.

Routine **Read** **Get 8 bit (provide the clock)**

out: DataIn read, transferred LSB first
mod: DataIn, C1

```
READ                             Read:
    MOVLW   8                        Move        #8,W
    MOVWF   C1                       Move        W,C1
L                                L$:
    CKON                             CkOn
    CKOFF                            CkOff
    BCF     3,0                      ClrC                    ;*
    BTFSC   5,BDATA                  TestSkip,BC   PortA:#bData    ;*
    BSF     3,0                      SetC                    ;*
    RRF     DATAIN                   RRC         DataIn
    DECFSZ  C1                       DecSkip,EQ  C1
    GOTO    L                        Jump        L$
    RETURN                           Ret
```

The three instructions marked with * can be replaced by one (RRC PortA,W) if the data bit input is wired on bit 0 of port A (i.e. bData=0)

Again, initializing the register with a one in high position, and waiting for that bit to be shifted out is as fast and saves a variable.

Routine **Read** **Get 8 bit**

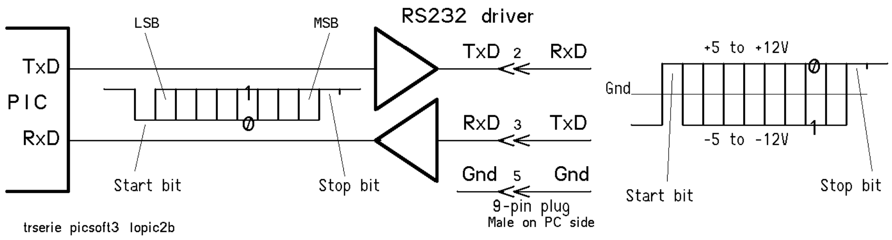
out: DataIn read, transferred LSB first
mod: DataIn

```
READ                             Read:
    MOVLW   B'10000000'              Move        #2'10000000,W
    MOVWF   DATAIN                   Move        W,DataIn
L                                L$:
    BCF     3,0                      CkOn
    BTFSC   5,BDATA                  CkOff
    BSF     3,0                      ClrC        ;*
    RRF     DATAIN                   TestSkip,BC   PortA:#bData    ;*
    BTFSC   3,0                      SetC        ;*
    RETURN                           RRC         DataIn      ; Carry out = 1 after 8 shifts
    GOTO    L                        Skip,CC
                                     Ret
                                     Jump        L$
```

9.3. RS232 serial transfers

Some 28-pins PICs have a built-in serial interfaces. By programming, it is easy to reach 9.6 kb/s and even 38kb/s with a 4 MHz processor. Synchronous programming has to be well understood for performing several tasks in parallel to the serial transfer.

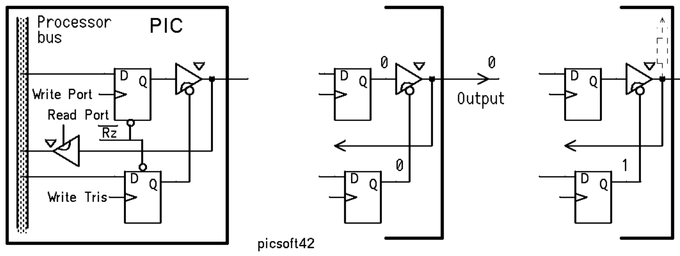
The serial interface toward a PC is shown in figure 1. A start bit is followed by the 8 data bits and 2 stop bits at least. For small applications, there is no need to bother with parity. The timer, studied later, is adequate for defining the bit period. For transmitting data, when the timer overflow, next bit is transmitted. For receiving data, the start bit is attentively checked. The timer for 1 and a half period is set (minus the response time), and then the signal is sampled every bit period. Adjustment of parameters and precision of the clock must be such that the last bit is sampled correctly.



More informations is available in French on pages www.didel.com/doc/

9.4. Bidirectional I/O and open-collector

Bidirectional signals are controlled by the "Tris" (tristate) direction registers. A "1" defines the corresponding pin as an input, a "0" as an output.



If the ouput controls an open-collector line, as e.g. with I^{2}C or 1-wire Dallas circuits, the data line is entirely controlled by the "Tris" register, since one need only write zeros. A zero is hence written on the port bit, and is issued on the bus when the corresponding tristate bit is clear.

One should be very careful if another bit of the same port is set or cleared. In this case the processor must read the port (it reads the 8 bits), modify the requested bit, and write the result back. This means that if our port is reading a one, that state will be copied in the output port. What is required in such a case is to work on a copy of the port, and use only write instructions to the port.

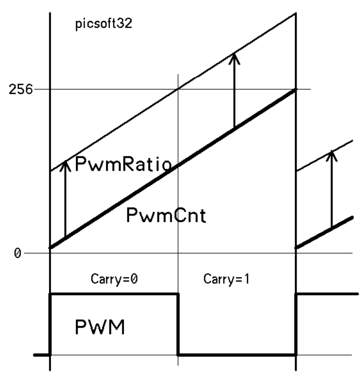
10. DC motor control

DC motors are available within a great range of size, cost, and quality (efficiency, life-time). The smallest ones are 6mm in diameter for a power of 0.1W. One-kilogram robots are quite happy with a 25mm 3 Watt motor, controlled by a 5V 1A amplifier (or 12V 0.5 A). On/off unidirectional control is as straightforward as lighting a lamp. Bidirectional control is implemented with 4 or 6 transistors, now available as miniature efficient ICs. Proportional control according to an analog value sent to a power amplifier is not efficient. Switching the motor on/off with an adequate duty cycle is the solution. Low cost motors have a high starting current which does not make them suitable for precise speed control. Good motors are unfortunatly rather expensive.

10.1. PWM

Pulse Width Modulation, or PWM, is a widely used scheme to reduce the average power transmitted to a lamp, resistor, or coil. The signal is active a percentage of the time, with a repetition rate compatible with the application. Frequency above 25 kHz avoids acoustic effects. Too low a frequency is not acceptable, especially with a low inertia stepping or synchronous motor. Hardwired PWM is available on many microcontrollers. Low resolution PWM is easy to program on the cheaper PICs, but there is always a problem with 0% or 100% duty cycles.

The simplest solution to generate PWM is to use an 8-bit counter which increments at a constant rate, and add an 8-bit value proportional to the PWM ratio. In order to reach 100% with value 255 (and not 255/256), the solution is to use a counter by 255. Adding the PwmRatio value to PwmCnt variable may generate Carry, directly related to the PWM value.



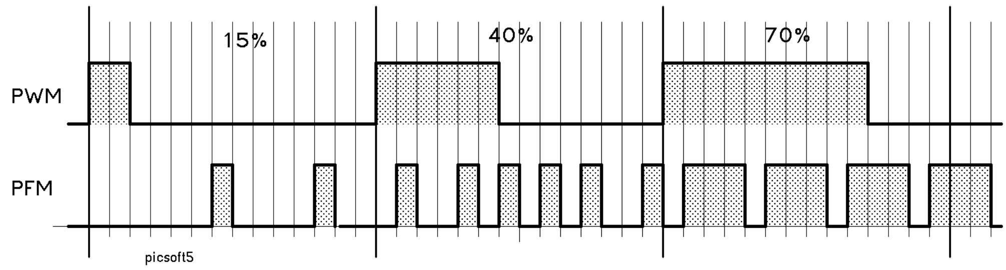
The corresponding program module is given below. If this program is executed continuously on a 16C86 at 4MHz, it takes at least 12 microseconds (i.e. the rate period is about 3 ms). A 330 Hz

PWM rate is a little slow, and an 8-bit resolution is clearly too high. The second program shows how to reduce the resolution, and increase the PWM speed: a divide by 15 counter and 16 PWM values are implemented by adding 16 at each cycle. The valid PWMRatio values are 0, 16'10, 16'20, .. 16'F0 (i.e. decimal values 0 to 15 followed by a SWAP instruction). This program is used in a synchronous loop, or called by a timer interrupt..

```
                                     8-bit PWM, < 330 Hz @ 4MHz
  INCF     PWMCNT                    Inc       PwmCnt   ;
  BTFSC    STATUS,2                  Skip,NE            ; Counter by 255
  INCF     PWMCNT                    Inc       PwmCnt   ;
  MOVF     PWMCNT,W                  Move      PwmCnt,W
  ADDWF    PWMRATIO,W                Add       PwmRatio,W
  BTFSC    STATUS,0                  Skip,CC
  MOTORON                            MotorOn   ; (macro Set PortB:#bMotor)
  BTFSS    STATUS,0                  Skip,CS
  MOTOROFF                           MotorOff  ; (macro Clr PortB:#bMotor)
  ...                                ...
                                     4-bit PWM, < 6 kHz @ 4MHz
  MOVLW    0x10                      Move      #16'10,W    ; PwmRatio   xxxx0000
  ADDWF    PWMCNT                    Add       W,PwmCnt
  BTFSC    STATUS,2                  Skip,NE
  ADDWF    PWMCNT                    Add       W,PwmCnt
  MOVF     PWMCNT,W                  Move      PwmCnt,W
  ADDWF    PWMRATIO,W                Add       PwmRatio,W
  BTFSC    STATUS,0                  Skip,CC
  MOTORON                            MotorOn
  BTFSS    STATUS,0                  Skip,CS
  MOTOROFF                           MotorOff
  ...                                ...
```

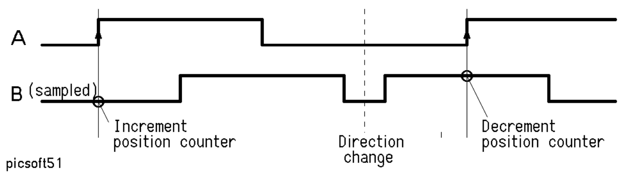
10.2. PFM

PFM is another way to get proportional control. Figure below explain the difference. The implementation is quite easy with a PIC, see www.didel.com/doc/DopiSmoo.pdf.



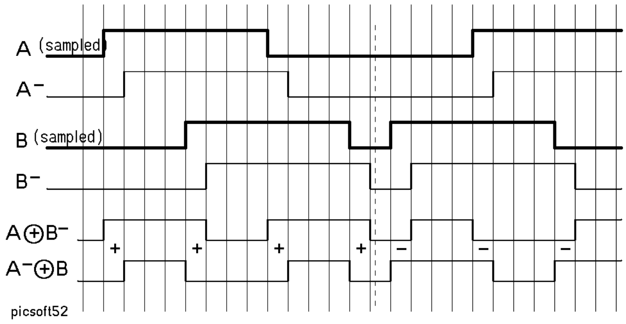
10.3. Rotary encoder

Rotary encoders, as found on computer mice are easy to decode if one just check for the edge on one channel, and increment or decrement the position counter according to the value sampled on the other channel. This scheme is quite convenient to implement with an interrupt on channel A, but there is problems in case of glitches, or small oscillations around the positive edge of channel A.



Sampling at high enough frequency A and B signals and applying the Sommer algortihm, as used in all Logitech mice, is reliable and needs a minimum amount of time. It is quite suitable for synchronous programming.

The principle is to keep the previous value A¯ and B¯, and compute the exclusive OR of combined signals. This is performed going through two level of tables,, the second table calling the routine to be performed when a step is done, usually incrementing or decrementing. It is a very efficient program, not so easy to understand.



picsoft52

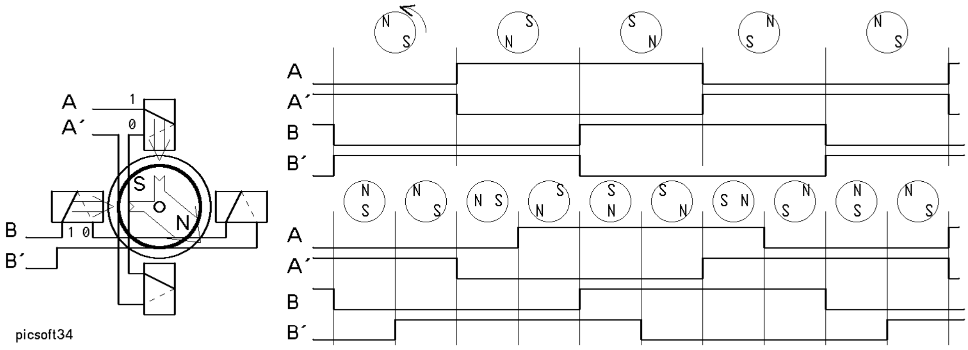
```
 This program implement the R.Som
; (Logitech Inc) algorithm on a PIC
; To be executed synchronously or
; lasts 12/13 us (single channel, 1
.macro          d     ; table element
     RetMove  #2´%1,W
.endmacro
; PortA input encoder
; PortB out counter low
.Loc DebVar
OldPort:        .16  1
Temp:           .16  1
CntLow:         .16  1 ; 16-bit cou
CntHigh:        .16  1
.Loc 0
Begin:
     Move      #2´11111111,W  ; i
     Move      W,TrisA
     Clr       W
     Move      W,TrisB   ; Outputs
     Move      PortA,W
     And       #2´11,W
     Move      W,OldPort
```

```
Loop:
     Move      OldPort,W
     Call      TaSwap
     Move      W,Temp
     Move      PortA,W   ; bits 1,0
     And       #2´11,W
     Move      W,OldPort
     Xor       Temp,W
     Call      TaJump
; Test: display value on PortB
     Move      CntLow,W
     Move      W,PortB
     Jump      Loop
TaSwap:
     Add       W,PCL
     d         00
     d         10
     d         01
     d         11
TaJump:
     Add       W,PCL
     Ret                 ; Nop
     Jump      I1        ; Increment
     Jump      D1        ; Decrement
     Ret                 ; Nop
I1:  Inc       CntLow
     Skip,NE
     Inc       CntHigh
     Ret
D1:  Dec       CntLow
     Skip,NE
     Dec       CntHigh
     Ret
.End
```

11. Stepping and synchronous motor control

11.1. 2-phase motor

Two phase stepping motors are available in many shapes and powers. The power to weight ratio is not very good, especially due to the fact that if the requested torque is too high, the motor will lose it synchronization and oscillate. The number of steps per turn depends on the windings and poles. In the example below, four coils (logically seen as 2 coils) provide 4 steps per turn, or 8 steps in the half-step mode.



picsoft34

Stepping motor controllers are available with a built-in up/down counter, decoder, and power amplifier. The cost of these cicuits is much higher than that of a PIC, which is easy to program to generate the required sequence. Recent miniature low dropout NMOS transistors or DC-motor bridges are directly controlled by the microcontroller outputs.

For instance, The program for a 4-phase motor is simply written

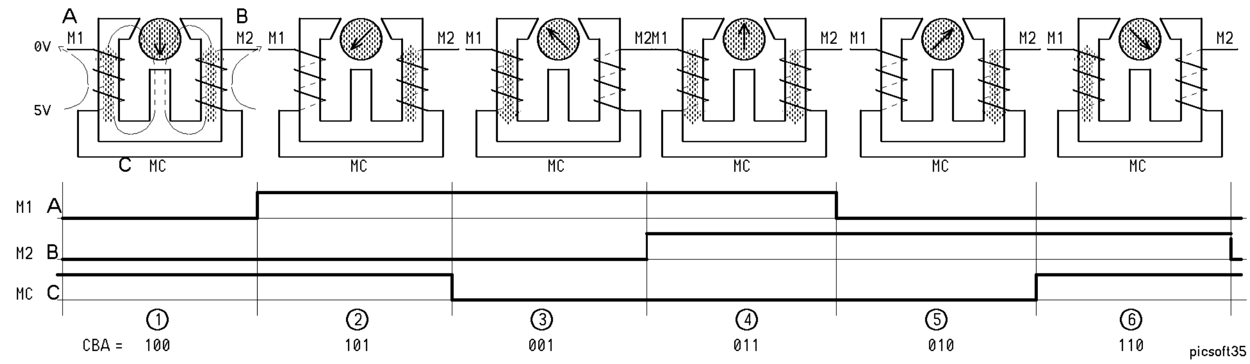
```
        Clr       VarStep
Loop:   Move      VarStep,W
        Inc       VarStep
        Call      DoStep
        Move      W,PortB
        ; delay according to rotation speed
        Jump      Loop

DoStep: Add       W,PCL
        DD        2'0101   ; A A' B B'
        DD        2'1001
        DD        2'1010
        Clr       VarStep
        DD        2'0110
```

11.2. 3-phase Lavet bidirectional motors

Watch motors are unidirectional. A trick in the magnetic circuit allow the motor to step always in the same direction with a single coil pulsed with a low duty cycle (i.e. a very low power). Programming such a motor is just a question of generating pulses of the correct duration. No such motor is commercially available, but it is easy to dismantle a Swatch or a kitchen clock and play with it. Due to the high resistance of the coil, a microcontroller output can be directly used.

Large (30mm in diameter) bidirectional watch motors are now available as the switec motor, including a 180:1 reduction gear. The dynamic torque is about 1mNm at 1 RPM. Max speed is 2 RPM.



The routine for making a complete turn and stop is easy to write. Variable "Step" is a pointer to the table. In the example below, the 3 wires of the motor are connected to portB, bits 2..0. No stop is required after the first three steps, as shown in the above figure, but a full 1ms is recommended before optionally asking for the next step, hence the 7 ms for one isolated turn.

Routine OneTurn and stop in a no-power state. Duration 7ms

```
ONETURN
      MOVLW     7
      MOVWF     STEP
A     MOVF      STEP,W
      CALL      TASTEP
      MOVWF     6
      CALL      ONEMSDELAY
      DECFSZ    STEP
      GOTO      A
      RETURN

TASTEP
      ADDWF     PCL
      RETLW     0
      RETLW     B'00000000'
      RETLW     B'00000101'
      RETLW     B'00000100'
      RETLW     B'00000110'
      RETLW     B'00000010'
      RETLW     B'00000011'
      RETLW     B'00000001'
```

```
OneTurn:
        Move      #7,W
        Move      W,Step          ; Point moteur position
  A$:   Move      Step,W
        Call      TaStep
        Move      W,PortB
        Call      OnemsDelay
        DecSkip,EQ  Step
        Jump      A$
        Ret

TaStep:
        Add       W,PCL
        RetMove   #0,W    ; never read
        RetMove   #2'00000000,W   ; Step=1, last step
        RetMove   #2'00000101,W
        RetMove   #2'00000100,W   ; M2 MC M1
        RetMove   #2'00000110,W
        RetMove   #2'00000010,W
        RetMove   #2'00000011,W
        RetMove   #2'00000001,W   ; Step=7, first step
```

Of course, in most applications, the steps of this program have to be executed within an interrupt routine triggered by a 1-ms timer (section 12.2). Bidirectional operations are also requires. The routine that executes a positive or negative step, according to a direction bit, is initialized by clearing the Step variable and can be written:

RIC324 JDN-DIDEL 17:46:59 30/11/03 8987 @TYPO:#MM0:JDN:@TYPO:#MM0:JDN:PICSOFT.TF 17

```
        TestSkip,BS   Var:#bStepDir
        Jump          Back
; Forward
        Move          Step,W
        Inc           Step
        Add           W,PCL
        bb            001       ; 0
        bb            011
        bb            010
        bb            110
        bb            100
        Clr           Step      ; 5, next is 0
        bb            101
Back:
        Dec           Step
        Inc           Step,W    ; test if -1
        Move          #5,0
        Skip,NE
        Move          W,Step
        Add           W,PCL
        bb            001       ; 0
        bb            011
        bb            010
        bb            110
        bb            100
        bb            101       ; 5
```

11.3. 3-phase smoovy motors

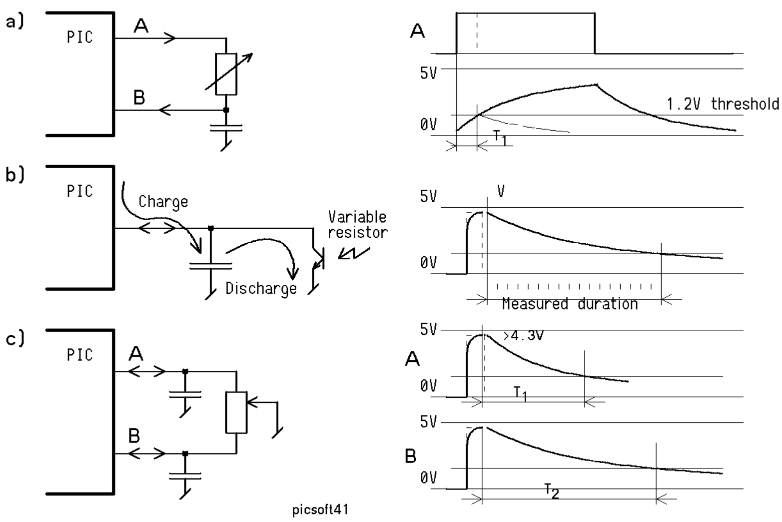
Synchronous motors need an angular sensor and a rather complex electronic in order to give their best performance. When miniaturization is concerned, synchronous 3-phase motors is the only way to reach small dimensions (down to 1.9mm) and high power/weight ratio. The smoovy motors are available in 3mm and 5mm diameter. Minimotor/Micromo proposes a 1.9mm motor controlled by a box 1000 times larger. Synchronous motors can be controlled as stepping motors, with the usual problems of stepping motors. A detailed presentation of open-loop control of the smoovy, including the efficient PFM scheme for a smooth rotation at any speed, is available in www.didel.com/doc/DopiSmoo.pdf.

12. Interfacing the analog world

When an A/D converter is not available, and an external serial I/O converter must be saved, several solutions allow conversion of voltage, current source, or resistor value into time or pulse trains, and usage of the timer or software loops to get an analog value equivalent.

The usual solution (fig a below) is to load a capacitor through the variable resistor to be measured. When the voltage on the capacitor is higher than the minimal one-level for the PIC input, about 1.2 Volts, the processor stops measuring time, and usually deactivates the A output to discharge the capacitor.

A better scheme is to use the bidirectionality of the ports to rapidly charge or discharge the capacitor. In figure b) below, the capacitors are loaded by a 2 microsecond pulse, and discharge in a few ms by the resistor. Two channels are preferably used for a potentiometer (fig c), in order to get a very strong symmetry over the span, and some independence of component value.



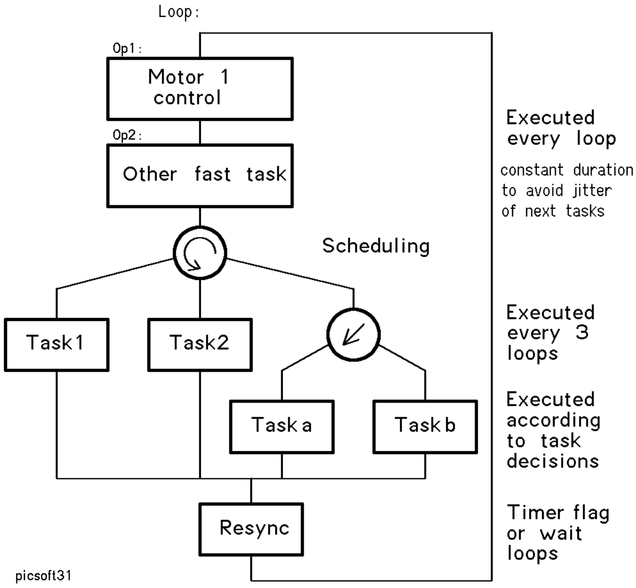
PWM or PFM signals followed by an R/C filter, or a better filter if required, can provide a low frequency analog output. This may be useful for some sensors, when a generated analog voltage has to be compared with a sensor value, in order to adapt some variable.

Recent PICs have all several 10-bits A/D converters. The need to do it by software has disappeared.

13. Multitask and real time

13.1. Synchronous programming

Synchronous programming means that all the operations the program has to do are selected within a loop of constant duration. Devices with precise timings are controlled every loop or every n loops. Other tasks are scheduled according to the previous task, to their priority, or to the raising of flags asking for operation. Resynchronization is performed by the timer if the processor has one that can be used for this. Otherwise, it is easy to add a wait loop, defined by a parameter adjusted for every task. No interrupts are allowed when synchronous programming is implemented. Interrupts are good only for low-performance real time, and can guarantee real-time precision on only one high-priority channel.



picsoft31

The loop is typically 100 µs or less, in order to service a high frequency communication or motor controlled task. This means the operation executed at every loop is 10 to 20 instructions (with a low speed 4MHz PIC). Switching between tasks takes several instructions. Long tasks have to be cut into pieces. It is easy to load the address of the next instruction to be executed into a register and go back to the loop. With the PIC having a single Work register, nothing has to be saved if the program is cut at the appropriate place. The scheduler will continue execution during the next loop.

Round-robin scheduling is programmed with a scheduling index incremented circularly at every loop. In the case of two tasks, one can write (SchIndex is cleared at start-up):

```
     MOVF     SCHINDEX,W               Move     SchIndex,W
     INCF     SCHINDEX                 Inc      SchIndex
     ADDWF    PCL                      Add      W,PCL
     GOTO     PRIOTASK1                Jump     PrioTask1
     GOTO     PRIOTASK2                Jump     PrioTask2
     CLRF     SCHINDEX                 Clr      SchIndex
LASTTASK                            LastTask:
```

With the low priority scheduler, the next task is prepared by the current one. It could be deemed a "please do this afterward" algorithm.

```
     MOVLW    0xFF&TASKA               Move     #16'FF.AND.TaskA,W    ; start-up initialization
     MOVWF    NEXTTASK                 Move     W,NextTask
...                                    ..
TASKA                               TaskA:
... do the work                        ... do the work
     MOVLW    0xFF&TASKB               Move     #16'FF.AND.TaskB,W
     MOVWF    NEXTTASK                 Move     W,NextTask
     GOTO     TASKDONE                 Jump     TaskDone
```

One must take care of the page where the scheduling loop and the tasks are. Adequate grouping is required.

13.2. Timer

The PIC timer can be used to guarantee that the synchronous loop is of constant duration. The timer is tested by a waiting loop, and reloaded when the loop duration is reached. No interrupt is required in this case, since the program has nothing else to do aside from waiting to begin the next loop iteration.

Test timer overflow flag and increment PortB when set

```
ORG 0                                  .Loc 0
     MOVLW   0                              Move     #0,W      ; Outputs
     TRIS    PortA                          Move     W,TrisA
     TRIS    PortB                          Move     W,TrisB
     MOVLW   B´00000111´                    Move     #2´00000111,W ; Prescaler :256
     OPTION                                 Move     W,Option
     CLRF    TMR0                           Clr      TMR0      ; First loop same length
LOOP                                   Loop:
     INCF    PORTB                          Inc      PortB     ; Every 256 x 256 us
W    BTFSS   0xB,2                      W$:TestSkip,BS  IntCon:#TOIF  ; test Timer Overflow Interrupt Flag
     GOTO    W                              Jump     W$
     BCF     0xB,2                          Clr      IntCon:#TOIF ; clear the flag
     GOTO    LOOP                           Jump     Loop
```

Most programs use the timer to start a regular interrupt in which all regular tasks must be performed, e.g. as seen previously, doing a motor step. When several tasks are controlled by interrupt, latency due to the termination of previous interrupt may be a problem.

Increment PortB by main program and increment PortA by Timer interrupt

```
     ORG     0                         .Loc 0
     GOTO    DEB                            Jump     Deb
                                       .Loc 4    ; Interrupt every 256 x 16 us, will increment PortA
     ORG     4                              ; Save F W in a typical application
     ; Save F W                             ; Reload the timer
     BCF     INTCON,2                       Clr      Intcon:#2 ;TOIF
     DECFSZ  CINT                           DecSkip,EQ   CInt
     GOTO    F                              Jump     F$
     INCF    PORTA                          Inc      PortA   ; Every 256 x 256 x 16 us = ~1s
F                                        F$:
     ; Restore W F                          ; Restore W F
     RETFIE                                 RetI
                                       ; Main program
DEB                                    Deb:
     MOVLW   0                              Move     #0,W            ; All outputs
     TRIS    5                              Move     W,TrisA
     TRIS    6                              Move     W,TrisB
     MOVLW   B´0000011´                     Move     #2´0000011,W ; Prescaler :16
     OPTION                                 Move     W,Option
     MOVLW   B´10100000´                    Move     #2´10100000,W ; GIE and TOIE on
     MOVWF   INTCON                         Move     W,IntCon
                                       Loop:         ; Main program increment PortB
LOOP                                        Inc      PortB
     INCF    PORTB                        ; Waiting loop 65 ms
                                        A$:DecSkip,EQ   C1
A    DECFSZ  C1                             Jump     A$
     GOTO    A                              DecSkip,EQ   C2
     DECFSZ  C2                             Jump     A$
     GOTO    M                              Jump     Loop
     GOTO    LOOP
```

14. Real time debugging

Debugging an application with real time constraints implies that only a few microseconds spy instructions can be inserted. See www.didel.com/picg/doc/AideDebug.pdf. The program below is not so useful, but it is a good example of tricky programming.

Measuring operation time cannot always be done by looking at the application signals. It is very useful to have at least one or two lines that can be used as outputs or inputs. An output can generate a synchronization pulse for the scope, or a pulse for measuring the duration of a loop, for estimating the jitter in an almost-synchronous program. The pulses can be counted by some external hardware, with the counter cleared or loaded by another output pulse. A pulse costs 2 microseconds. In order to be able to estimate more easily the number of pulses on the scope, the following debug module can be used (S1On and S1Off are macros, see section 5.6). Une pulse is removed every 4, and every 64 pulses, the pulse lasts one full period. These numbers can be other powers of two.

Visualize event duration on scope – takes 7us

```
     MOVLW   4                              Move     #4,W
     ADDWF   CYCNT                          Add      W,CyCnt  ; An auxiliary variable.
     BTFSS   STATUS,2                       Skip,EQ
     BTFSS   STATUS,1                       Skip,DS           ; Remove 1 pulse every 4
     S1ON                                   S1On              ; Activate test pin
     BTFSS   STATUS,2                       Skip,EQ           ; Longer pulse every 64
     S1OFF                                  S1Off             ; De-activate test pin
```

Using a serial line (clock and data) for transfering variables to be checked or modified is slow, but very informative. Testing two switches in order to increment or decrement a given variable value takes 5 to 10 microseconds and may be convenient in some cases.

The CALM assemblers have been developed by Patrick Faeh while at LAMI-EPFL. PIC modules have been adapted by Johann Rohner. The SMILE-NG editor-assembler has been developed by Sebastian Gerlach when he was undergraduate student at EPFL. Picolo is a new environment similar to SmileNG, running on Linux, Apple and Windows; it has been developed by Fabien Zennaro and Gilles Dubochet. A translator Microchip-CALM is being developed by Kaspar Schiesser.