CH-1092 Belmont, Switzerland
info@didel.com, www.didel.com

Fichier www.didel.com/doc/DopiSync.pdf

Synchronous programming

for the 16F84/16C5x/12C50x microcontrolers

1. Introduction

Our objective is to learn to program the PIC microcontrollers for applications with strong real time constraints and miniaturization requirements.

We focus on the 16F84 as a convenient processor for developping the software, due to its flash program memory and the convenience of Smile-NG. But the objective is to do applications with the smaller 16C5x (available in a 5mm x 6mm SSOP20 package) and with the 8-pin 12C50x (available in a 5mm x 5mm SO8 package).

Porting a debugged program from the 16F84 toward the 16C5x (with almost compatible I/O) or toward the 12C50x (with only six I/O) is made easy if the differences between these processors are understood from the beginning.

Synchronous programming is a very efficient technique that garantees heavy real time constraints with low speed (that is low power) processors. It is compatible with the 16C5x/12C50x processors which do not have interrupts.

Interrupt handling is not so convenient with the PIC microprocessor, and when several interrupts are simultaneous, real time constraints may not be satisfied. If fast communications are required, one have to use the more sophisticated PICs, with serial, SPI, I^2C interfaces supported by interrupt. We are not concerned here with these applications; we are interested with networks of small PICs, but we will develop communication schemes appropriate with 8-pin PICs, and use if required the more sophisticated PICs as a concentrator toward a PC or other major processor.

2. Features of the PIC 16F84/16C5x/12C50x

Microchip data sheet documents all the features of these processors. We will concentrate here on what we need for synchronous programming and on the differences between the PIC 16F84/16C5x/12C50x. The reader is supposed to be familiar with PIC programming and CALM notations, as explained for instance in the **PIC***Smart* brochure.

2.1. Instruction differences

Two Add/Sub instructions of the PIC 16F84 are not supported by the 16C5x/12C50x. The **PIC***Smart* explain how easy it is to avoid their use.

```
Add     W,#Val,W
Sub     W,#Val,W
```

The Ret instruction (return from routine) is not supported. It must be replaced by

```
Ret     #Val,W   (as used for accessing tables)
```

Register W is modified; this means that the 16F84 routines must not pass a parameter back in register W.

On the 16C5x/12C50x, it is easy to define a macro and be compatible (assembler may do it automatically)

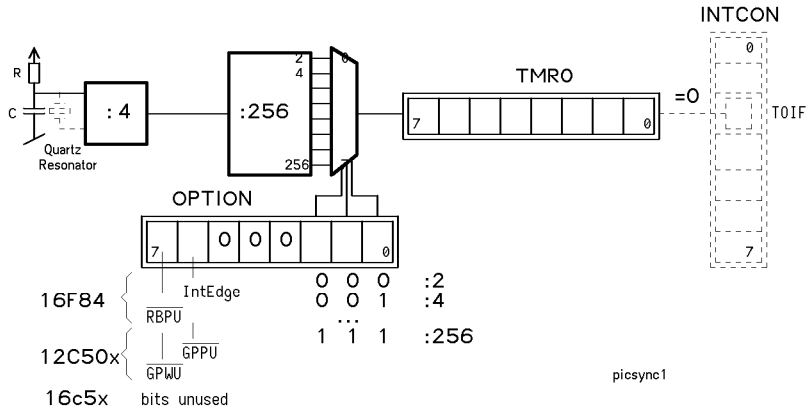
```
.Macro  Ret
        Retmove #0,W
.Endmacro
```

It will be seen that we do not use many routines. Most routines we could be tempted to create are short, and not called from many places. They loose time (4 microseconds for a Call and Return) and we do not really need to save memory space. On the Pic 16C5x/12C50x, only two levels of routines are allowed; since tables are fequently used and are implemented as routine calls, we have to program the 16F84 with the idea we can use only one level of routines.

2.2. Timer

Since we will use the PIC timer to synchronize the task evolution, we cannot use the external clock input to count events and we will ignore for the moment the external clock options and interrupts.

Our model of the timer is given in Figure 1.



The processor clock (quartz or RC network) is divided by 4 to give the 1 MHz frequency that defines the 1 microsecond instruction time that will always be our reference. A programmable predivider is implemented before the TMR0 counter that can be read and written by the processor.

The three low bits of OPTION registers control the predivider. Bits <5..3> are zero in the timer configuration. Bits <7,6> will be explained later.

On the 16F84, the TMR0 counter overflow sets the TOIF flag in register INTCON. This flag can be cleared with a "Clr Intcon:#TOIF" instruction.

Since the TMR0 is a counter, we have to initialize it with the number of increments to overflow. Preparing a delay of 100 microsecond for instance, with the predivider by 2 is programmed as:

```
; Initialization
      Move      #2'00000000,W       ; Predivide by
      Move      W,Option
      Move      #256-(100/2),W      ; 100 µs
      Move      W,TMR0      ;  (1)
      . . . .
```

```
; Inside program loop, when synchronization is required
W$:   TestSkip,BS   Intcon:#TOIF
      Jump          W$      ; Wait until TOIF bit is set (2
      Move          #256-(100/2),W ; re-init timer dur.
      Move          W,TMR0        ; (1)
      ;---Continue
```

From point (1) to point (2), the time is indeed 102 µs, due to the timer initialisation and the Jump W instruction that may loose a cycle. One can correct this by writing Move #256-(100/2)-2, W. Keep this expression in the source program: the assembler will calculate it without mistake, and the expression makes our intention quite clear to the reader, that is a good documentation practice.

But we will not use the above initialization and waiting loop. The PIC 16C5x/12C50x do not have an INTCON register and the TOIF flag, since it does not support interrupts. The only way to know that the timer has reached a zero state is to test it.

```
W$:     Move      TMR0,W
        Skip,EQ
        Jump      W$
```

This is 3-instruction loop and if the predivider divides by two, there is a risk that the "Move TMR0,W" instruction that sets the EQ/NE flag misses the zero value. At first pass for instance, TMR0 value may be 1, during Jump W instruction it is zero, and at the next "Move TMR0,W" instruction, its value is 255 !

We have to write the previous program module as:

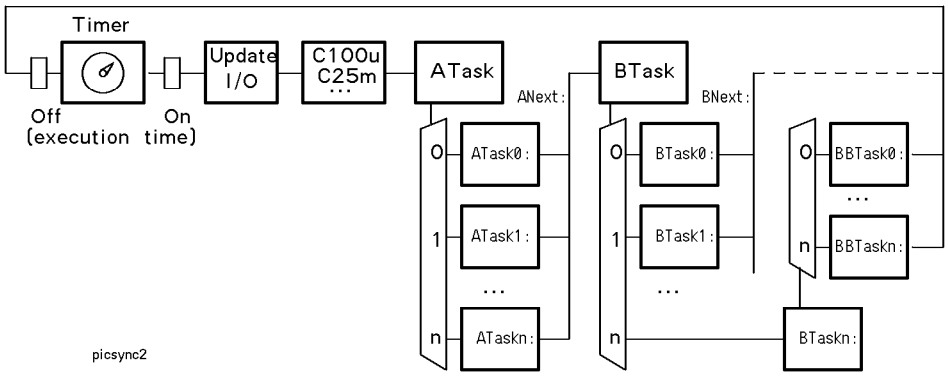
```
; Initialization
      Move      #2'001,W          ; Predivide by
      Move      W,Option
      Move      #256-(100/4),W        ; 100 µ
      Move      W,TMR0
      . . . .
```

```
; Inside program loop
      . . . .
W$:   Move      TMR0,W
      Skip,EQ
      Jump      $W        ; Wait until TMR0 is zero
      Move      #256-(100/4),W   ; Reinit timer
      Move      W,TMR0
      ;---Continue
```

Do not be tempted to use the Test TMR0 (Move TMR0,TMR0) instruction instead of the Move TMR0,W instruction. It does not work in this case!

3. Synchronous programming

The general scheme for synchronous programming is given in Figure 2.



The timer synchronizes on a fixed period basis. There should be some waiting time before resynchronisation: the best way to measure it is to clear a free bit on a port before starting the wait for the timer and set it when the timer is reinitialized. A scope will clearly show the execution time every 100 µs. There will be an important jitter if the subtasks are not of similar length. If the time is too long, the time will miss one turn, and this is easy to recognize on the scope.

I/O update is usually the first thing to do after synchronization, in order to avoid some jitter which could be listen or disturb a stepping motor or a PWM ratio. Port bits usually control different actions and are updated by different tasks. Updating is made by the different tasks in a register, copied to the port before the tasks introduce their jitter.

Updating time counters is usually the next task. As it can be noticed in our example, a task is longer when there is a next task to be prepared. If too many instructions, it can be split in two subtasks executed on two consecutive 100 µs phase.

Synchronous programming has some similarity with the "virtual peripheral" concept of Scenix, but the implementation is quite different. Let us see several examples and build our library of tasks handling.

4. Programming techniques for synchronous programming

4.1. Time counters

Long times can be measured with counters incremented every phase. C100u is our first counter, incremented every 100 µs. As an 8-bit counter, it counts by 256 and overflow every 25.6 ms. We can test the zero to increment a C25m counter every 25 ms, that will overflow after 6.3 sec.

The instruction we have to write are:

```
        IncSkip,EQ  C100u
        Jump    Nxt
        IncKip,EQ  C25m
Nxt:
```

If we need within a task to measure a duration, we can define a variable "Time" do as for the HC11 timer:

```
; At begin of task:
        Move    C25m,W
        Move    W,Time
; At end of task:
        Move    C25m,W
        Sub     Time,W
                ; result in W is the time duration in ms (max. 6.3 s).
```

If we need long delays, a 1 sec counter C1s going up to 256 s may be preferable to a 6.3 s counter. It is easy to implement with a C25m counter that counts by 40 (and not 256). The trick valid for divide by 10 counters (**PIC***Smart* section xx) is applied here for the divide by 40 counter.

```
        IncSkip,EQ  C100u
        Jump    Nxt
        TestSkip,BC C25m:#2    ; Test if 10100 = 32+8
        TestSkip,BS C25m:#4    ; in order to divide by 40
        Jump    Nxt
        Inc     C1s                ; 1 second counter
Nxt:
```

The 1 second period is not very precise. We can adjust within a 4% precision changin

```
        Inc     C100u
        Move    #250,W
        Xor     C100u,W            ; Compare for equality
        Skip,EQ
        Jump    Nxt
        Clr     C100u
```

It is more efficient to use a down counter:

```
        Move    #250,W
        DecSkip,EQ  C100u
        Jump    Nxt
        Move    W,C100u
```

4.2. Split tasks

A typical task, for instance blinking a LED in response to an external event, supposes to check for an event and then blink a certain number of times with the correct duration. If the processor has nothing else to do, it is easy to program with waiting loops, as explained in Picsmart.

Now, if we want to do other tasks in parallel, we can split the work and execute it every 100 μs, for a duration up to 20 μs for instance (max 20 instructions). This leaves the space for 4 other tasks that will also be served every 100 μs.

The first task, wait for a flag active, will use only 3 instructions. Blinking the led is a slow process. A LED duration counter will decide how many times 100 μs the LED is ON and how many times the LED is OFF. Another counter will count the blinks. Hence, the subtasks to be executed every 100 μs phase are the following.

```
BTask0:     Test the flag.
                If zero,  continue
                If one,   LED on, prepare LED on duration counter and number of blinks counter, switch to BTask
BTask1:     Test LED duration counter
                If ><0, continue
                If =0, LED off, prepare LED off duration, switch to BTask2, continue
BTask2:     Test LED duration counter
                If ><0, continue
                If = 0, decrement number of blink counter
                        If = 0, switch to BTask0, continue
                        If ≠ 0, LED ON, prepare LED on duration, switch to BTask1, continue
```

4.3. Selecting the subtask

It is specially easy with the PIC to select the task to be executed. A task pointer BTask is initialised to zero and can then be incremented/decremented/cleared at the end of a subtask, so the new subtask will be executed at the next 100 µs phase.

The switch is written

```
Move    BTask,W
Add     W,PCL
Jump    BTask0
Jump    BTask1
Jump    BTask2
```

This takes 4 microseconds for any number of tasks.

5. Blinking program

Let us write and test a synchronous program that blinks 4 times a LED every 3 seconds. The "interrupt" every 3 seconds is provided by a Bspace decounter that is decremented when C25m is incremented, every 100 µs. Bspace is initialized at 3000ms/25ms= 120 and sets a flag when it reaches zero. There is no need for other subtasks and we will save the 4 microsecond of the task switch and the task variable.

5.1. Final program

The complete program (next page) must document the port used for the LED, the variables and the modules. Parameters are defined in the top of the listing, so they are more easily changed to test their effects. Their preferred place is next to the variables they are related to.

There are several comments to give about this program. The bBlink flag is indeed not required. It is here to show the general scheme of activating a flag in one task, testing it and clearing it in another task. A simpler solution would be to have a BTask00 task, with only a Jump BNext instruction, add a BTask01 task, with the preparation of the blink. When the 3 sec are over, it is enough to increment BTask to start the blinking operations 100 µs later.

CLedOn and CLedOff counters are not used at the same time; a single CLed counter is enough. CBlink counter could also be mapped on the same variable, but this would mean that the space between two set of blinks will be constant, and not the period of the set of blinks.

One can notice that macros have been defined for I/O actions. It makes more easy the transfer from one application to another, from one processor to another, if the I/O ports are not explicitely mentionned within the program.

Program **Picgs5.asm** Blink 4 times every 3sec

```
.Title picgs5 jdn 24.7.99
.proc 16F84

Constant Parameters
ICLedOn     = 100/25 ; 0.1s
ICLedOff    = 250/25 ; 0.25s
ICBlink     = 4      ; Number of blinks
ICSpace     = 3000/25 ; every 3s

Variables Registers
            .Loc     16'C
C100u:      .16      1        ; Counter 10
C25m:       .16      1        ; Counter 25
BTask:      .16      1        ; Blink task
CLedOn:     .16      1        ; LED on cou
CLedOff:    .16      1        ; LED off
CBlink:     .16      1        ; Number of
Cspace:     .16      1        ; Space count
Flag: .16   1        ;Flag reg for ut to 8 fla
 bBlink     = 0

Variables Timer
ICOption    = 2'00000001 ; Div b
ICTimer     = 256-(100/4)+1  ; 100 µs

Variables PortB LEDs
bLed  = 0       ; Blinking LED on RB0
MDirB = 2'0000000       ; Outputs
InitB = 2'11111111      ; LED off
.macro          LedOff
      Set       PortB:#bLed
.endmacro
.macro          LedOn
      Clr       PortB:#bLed
.endmacro

Variables PortA Exec time on RA0
bS0   = 0       ; Syncho oscillo
MDirA = 2'11110         ; RA0 out
InitA = 0
.macro          S0Off
      Clr       PortA:#bS0
.endmacro
.macro          S0On
      Set       PortA:#bS0
.endmacro

Program Initialization  .Loc 0
      Move      #MDirA,W
      Move      W,TrisA
      Move      #InitA,W
      Move      W,PortA
      Move      #MDirB,W
      Move      W,TrisB
      Move      #InitB,W
      Move      W,PortB           ; Leds all off
      Clr       C100u
      Clr       C25m
      Move      #ICOption,W   ; Prescaler :4
      Move      W,Option
      Move      #ICTimer,W    ; 100 us
      Move      W,TMR0
;  Init BTask
      Clr       BTask
;  Init STask (action every 3 sec)
      Move      #ICSpace,W
      Move      W,CSpace

Program 100 us loop
Loop:           ; Cycle 100 us
S0Off
 W$:  Move      TMR0,W
      Skip,EQ
      Jump      W$
      Move      #ICTimer,W    ; restart for 100 us
      Move      W,TMR0
S0On
      IncSkip,EQ  C100u
      Jump      Next
; Every 25 ms
      Inc       C25m
      Dec       CLedOn
      Dec       CLedOff
      Dec       CSpace
 Next:

Module STask Every 3 sec, ask for a bink sequence
STask:
      Test      CSpace
      Skip,EQ
      Jump      SNext
; Reinint CSpace delay and prepare for Blink task
      Move      #ICSpace,W
      Move      W,CSpace
      Set       Flag:#bBlink
 SNext:

Module BTask Blink n times
      Move      BTask,W
      Add       W,PCL
      Jump      BTask0
      Jump      BTask1
      Jump      BTask2
 BNext:
; No more task to execute
      Jump      Loop

Module BTaskn Blink a Led (parameters in the beginning)
BTask0:             Wait for blink flag
      TestSkip,BS   Flag:#bBlink
      Jump      BNext
;Prepare for the blinking
      Clr       Flag:#bBlink
      Move      #ICBlink,W
      Move      W,CBlink
      Move      #ICLedOn,W
      Move      W,CLedOn
      LedOn
      Inc       BTask
      Jump      BNext
BTask1:             Keep Led ON
      Test      CLedOn
      Skip,EQ
      Jump      BNext
;  Initialize for LED off
      Move      #ICLedOff,W
      Move      W,CLedOff
      LedOff
      Inc       BTask
      Jump      BNext
BTask2:             Test if finished, blink again if n
      Test      CLedOff
      Skip,EQ
      Jump      BNext
; Test if good number of blinks
      DecSkip,EQ  CBlink
      Jump      BT2
; Yes, finished
      Clr       BTask
      Jump      BNext
; No, reinit one blink
 BT2: Move      #ICLedOn,W
      Move      W,CLedOn
      LedOn
      Dec       BTask
      Jump      BNext
.End
```

JDN 1999