dsPIC Programming
Beginners Guide

Pedro Costa, Powertrain Team Leader, Projecto FST *

2014

*Reviewed by Electronic Team Leader Daniel Pinho

Contents

1 Introduction

This guide serves as a complement for the dsPIC Programming workshop, and his purpose is to facilitate the acknowledgment process involved in programming for a dsPIC Microcontroller. Note that this guide does not invalid the use of the reference manuals provided by Microchip©, those being:

- dsPIC30F Family Reference Manual
- dsPIC30F6012A Reference Manual
- dsPIC30F4013 Reference Manual

Still this guide pretends to cover the basic topics of the modules involved. The document is therefore divided, after a small system introduction, by each peripheral module, where a brief explanation of their functioning principles and basic configuration steps/process takes place.
As a final introduction note I reinforce that this guide only suits for the beginning process and the references plus application notes provided by Microchip© should always be the guide for any program developed. Further improvement from experienced dsPIC programmers may accrue from the compiler manual as well, that said I leave here the manual for *xc16* compiler.

- XC16 C Compiler Users Guide

2 dsPIC

2.1 What is a dsPIC?

Using Microchip© definition:
"A Digital Signal Controller (DSC) is a single-chip, embedded controller that seamlessly integrates the control attributes of a Microcontroller (MCU) with the computation and throughput capabilities of a Digital Signal Processor (DSP) in a single core."
So in other words a dsPIC allows not only the use of a Microcontroller capability's but empowers it with a Digital Signal Processor. This raises two questions, what is a Microcontroller (MCU) and what is a Digital Signal Processor(DSP)? The second one will not be analyzed in this document, which leaves...

2.2 What is a Microcontrolelr?

A Microcontroller is basically a Microprocessor with a variety of peripherals integrated in the same IC. Answering the question what is a Microprocessor will not be covered in this guide since its intuit is not a computer architecture overview. This union between the processor and the peripherals allows a enormous power of parallel tasking where each peripheral can function simultaneously with the processor, allowing the programmer to take advantage of the computation power at is reach. In order for peripherals to work simultaneously to the processor there must be a mean (or more) that allows an interaction between both the systems, one of this means is Interrupts and will be approached later on.

2.3 Peripherals

The dsPIC offers a wide variety of peripherals namely:

- Timers
- ADC
- CAN
- SPI
- I2C
- UART
- Input Compare

- Change Notification
- Output Compare
- Data Converter Interface
- EEPROM Memory

These peripherals will facilitate your job as a programmer in any of the aspects each peripheral, covers as well as adding efficiency to the task being performed. Each peripheral has several modes of operation which make them highly versatile, and capable of interacting with almost anything.

3 Concept Introduction

When dealing with dsPIC's there are several concepts and notations that are required by the programmer to properly handle and configure the several dsPIC's properties.

3.1 Oscillators

First concept we are going to introduce is "Oscillation". For any Microcontroller to operate there is the need of setting the pace so that a flow of instructions can be properly executed. The system's that ensure this operation are the oscillators. dsPIC module can work either with an internal oscillator or an external one, note that the internal oscillator is not as stable as an external one (e.g. Crystal) and therefore functions at lower frequencies.
This introduces F_{osc} which is the frequency of oscillation (dsPIC has the ability to multiply or divide a frequency of an external oscillator achieving higher or lower frequencies than the one externally provided, still F_{osc} is the end frequency, from which dsPIC generates it pace). So F_{osc} is the frequency of oscillation but it is not the frequency of each cycle and from here we denote F_{cy} as the frequency of each cycle. The family reference states that

$$F_{cy} = F_{osc}/4 \tag{1}$$

The fact that dsPIC uses 4 oscillation cycles to perform one instruction is related to his internal architecture and therefore we will not develop this issue. What is important to understand is the relation between F_{cy} and F_{osc} because it will be of the most relevance for almost every peripheral the dsPIC provides.
Another relevant fact is that we can manipulate the input oscillation frequency in order to achieve multiples or sub-multiples of this frequency and use it as F_{osc}. We can do that through some configuration registers. As this functionality is not of the major importance for beginning process we will not cover it in this work.

3.2 Registers

Register manipulation is a major part of dsPIC, or any other Microcontroller, programming. A Register is nothing more than a segment of memory with a given number of positions.
The number of positions on the registers varies with the Microcontroller type, for our particular case this number is 16. This means that each register will have 16 positions which can either be 1 or 0.

So in order to configure any module in the dsPIC it is necessary to properly set all the necessary values on all the registers concerning the modules we want to program. For example, if we need to configure the UART module, we must set all the UART control registers. This particular registers can be found in page 489 of the dsPIC30F Family Reference.
Still to have in code access to the registers by their names we have to include a library that provides all the headers we need, and so we get the first code line of our projects supposing we are using dsPIC30F6012A:

```
1  #include <p30F6012A.h>
2
```

This concept is the foundation of dsPIC programming, all functionalities dsPIC may offer are accessible, controlled and manipulated through this registers, and in order to implement any program in the dsPIC modules you must have a clear understanding of how to manipulate this registers.

3.2.1 Registers Manipulation

Lets take for example The following Register:

Register 2-1: SR: CPU Status Register

Upper Byte:							
R-0	R-0	R/C-0	R/C-0	R-0	R/C-0	R -0	R/W-0
OA	OB	SA	SB	OAB	SAB	DA	DC
bit 15							bit 8

Lower Byte: (SRL)							
R/W-0(2)	R/W-0(2)	R/W-0(2)	R-0	R/W-0	R/W-0	R/W-0	R/W-0
IPL<2:0>			RA	N	OV	Z	C
bit 7							bit 0

Figure 1: Register Example

Registers in dsPIC Family Reference Manual will always have a similar representation to the one in the figure above. The register name is located on the top left corner, this case the register is called *SR*. Notice that the register has 16 bits numbered from 0 to 15, and each bit or groups of bits have their one identifier. The bit/bits in the registers can be of 3 major types:

- Control bits. Where you can control a certain parameter.
- Status bits. Provide us information on certain parameters.[1]
- Buffer bits. Provide buffing space[2].

For example IPL<2:0>bits present in the register displayed in figure 1 let us control the CPU Interrupt Priority Level, and OA bit inform us if there was an Overflow in Accumulator A. If we have made the proper library inclusion we can now access any bit in any register with the following notation, <Register Name>bits.<Bit/Bits Name>, for example

```
1       SRbits.IPL = 3; //Setting the CPU Interrupt Priority Level to 3;
2       if(SRbits.OA == 1){
3         someting();  //Accumulator A overflowed
4       }else{
5         someting_2();//Accumulator A did not overflowed
6       }
7
```

This was the first example of registers read/writes. It is by a similar process that you can configure almost every parameter of the dsPIC.

[1]Some bits can provide status and control at the same time.

[2]Typically this registers are present in digital communication peripherals to store messages to send or received ones

4 Interrupts

Interruptions are the main mean of interaction between the processor and the peripherals. "Basically, an interrupt is exactly defined by it's title", they interrupt the normal instruction flow of the processor and diverge it to a new focus point associated with a given event. So basically interrupts are generated by hardware events, so that the processor has the ability to handle them. When discussing interruption a major point that needs attention are priorities.

4.1 Priorities

Priorities are necessary in an environment where several events that can trigger interrupts, there maybe cases where we don't want to be interrupted and cases where we want to give more relevance to some interruption over others.
To manage this dsPIC as a set of 14 priority levels where level 14 is the most priority level. From those 14 only 7 are user selectable, this means that the upper priority levels (8-14) are reserved for processor exceptions and traps. Still we can distribute our interruptions in 7 levels of priorities, and when configuring several interrupts in a program we must take into consideration their priorities so that everything can work as expected. Not only the peripherals interrupts have priorities, CPU has its one priority and can also be selected by the user selectable priorities, this is important so that you can make sure certain "critical" parts of your code don't get interrupted.

4.2 Interrupt Vector Table

Suppose now that two different interrupts with the same attributed priority arrive at the exact same time, in order to keep th code flow running dsPIC has 2 Interrupt Vector Tables in which all the interrupts are listed in a given order so if two different interrupts with the same attributed priority arrive at the exact same time the one upper in the selected table will execute first. The 2 Tables are Interrupt Vector Table (IVT) and Alternate Interrupt Vector Table (AIVT), they are practically equal the difference is that "AIVT supports emulation and debugging efforts by providing a means to swhich between an application and a support environment without requiring the interrupt vectors to be reprogrammed".

4.3 Traps

Traps are none-maskable (which means they cannot be disabled), nestable interrupts which have a fixed priority structure. Their purpose is to provide the user a mean to correct erroneous operation during debug and when operating within the application. The dsPIC30F has four implemented sources of non-maskable traps:

- Oscillator Failure Trap
- Stack Error Trap
- Address Error Trap
- Arithmetic Error Trap

4.4 Nesting

Nesting is the ability to interrupt, a interruption routine that is in progress, if and only if the new interruption has a higher priority level than the one in progress. Nesting is by default enabled, it can be disabled using $NSTDIS$ in $INTCON1 < 15 >$.

```
1       INTCON1bits.NSTDIS = 0; //Enable Interrupt Nesting
2       INTCON1bits.NSTDIS = 1; //Dissable Interrupt Nesting
3
```

4.5 Interrupt Control and Status Registers

The following registers are associated with the interrupt controller[3]:

- **INTCON1, INTCON2 registers**
  Global interrupt control functions are derived from these two registers. INTCON1 contains the Interrupt Nesting Disable (NSTDIS) bit, as well as the control and status flags for the processor trap sources. The INTCON2 register controls the external interrupt request signal behavior and the use of the alternate vector table.
- **IFSx: Interrupt Flag Status Registers**
  All interrupt request flags are maintained in the IFSx registers, where x denotes the register number. Each source of interrupt has a Status bit, which is set by the respective peripherals or external signal and is cleared via software.

[3]This section is identical at section 6.4 of the Family Reference Manual

- **IECx: Interrupt Enable Control Registers**
  All Interrupt Enable Control bits are maintained in the IECx registers, where x denotes the register number. These control bits are used to individually enable interrupts from the peripherals or external signals.

- **IPCx: Interrupt Priority Control Registers**
  Each user interrupt source can be assigned to one of eight priority levels. The IPC registers are used to set the interrupt priority level for each source of interrupt.

- **SR: CPU Status Register**
  The SR is not specifically part of the interrupt controller hardware, but it contains the IPL<2:0>Status bits (SR<7:5>) that indicate the current CPU priority level. The user may change the current CPU priority level by writing to the IPL bits.

- **CORCON: Core Control Register**
  The CORCON is not specifically part of the interrupt controller hardware, but it contains the IPL3 Status bit which indicates the current CPU priority level. IPL3 is a Read Only bit so that trap events cannot be masked by the user software.

If Nesting is disabled then only one interruption will occur each time and interruptions that arrive while in a interruption execution, no matter the priority, will wait for the end of the first.

4.6 Interrupt Functions

You might wonder where to declare an interrupt function since you will never make a call on that function, you can virtually place it anywhere as long as it is within a file that the compiler actually compiles. Interruption Headers are very similar, some examples:

```
1       \*Timer1 Interruption*\
2       void __attribute__((interrupt, auto_psv, shadow)) _T1Interrupt(void);
3       \*Timer2 Interruption*\
4       void __attribute__((interrupt, auto_psv, shadow)) _T2Interrupt(void);
5       \*CAN2 Interruption*\
6       void __attribute__((interrupt, auto_psv)) _C2Interrupt(void);
7       \*Input Capture 1 Interruption*\
8       void __attribute__((interrupt, auto_psv)) _IC1Interrupt (void);
9       \*UART2 Recive Interruption*\
10      void __attribute__((interrupt, auto_psv)) _U2RXInterrupt(void);
11
12
```

Using this headers you can now declare your interruptions and be assured that if properly configured the interruption code will execute when the interruption event comes. Inside the interruption you must clear the interruption flag in order to ensure that the interruption will occur again. So you interruption should look something like this:

```
1       \*Timer1 Interruption*\
2         \*Some Code*\
3         IFS0bits.T1IF = 0;     /* clears interruption flag */
4       }
5
```

5 Timers

5.1 Introduction

Timers offer a time count by increasing the value on a specific register with a given frequency. Timers are one of the most important modules of the dsPIC and it is crucial to master their functionalities in order to use the full potential of this module. There are 3 types of timers, Type A, Type B and Type C, still they are quite identical and only diverge in specific applications and properties.

5.2 Timer Mode

Timer mode is the only mode we are going to approach were, still if you master this mode it will be quite easy to use the other ones.
Timer mode is the simplest mode of operation and it is supported by all 3 types of timers. In the timer mode the input clock to the timer will be $F_{osc/4}$ this means that the input clock is F_{cy} as we have seen in the Oscillator section. When enabled, Timer will increment based on the instruction cycle and a prescaler.
The Register that the timer increments is *TMRx*, where x equals the timer module you are using (dsPIC6012A has 5 timer modules available), you can read the *TMRx* anytime, "a read of the TMRx register does not prevent the timer from incrementing during the same instruction cycle". The second relevant register is *PRx*, this register provides the timer with a period and so it is called period register, in fact what happens is that when the value in *TMRx* matches the value in *PRx*, *TMRx* resets the counting meaning:

$$PRx = TMRx \implies TMRx = 0 \tag{2}$$

If the Timer interruption is enabled then is also at the condition mention above that the interrupt occurs.

5.3 Timer Prescale

Timer Module offers a settable prescale for the input clock that can be:

- 1:256 prescale value,meaning $F_{Tx} = F_{cy}/256$
- 1:64 prescale value,meaning $F_{Tx} = F_{cy}/64$
- 1:8 prescale value,meaning $F_{Tx} = F_{cy}/8$
- 1:1 prescale value,meaning $F_{Tx} = F_{cy}/1$

F_{Tx} will be the frequency on which timer module increments the $TMRx$ value.

You can now easily choose the value for PRx to achieve a given Period T,

$$PRx = \frac{T * F_{cy}}{TxCONbits.TCKPS} \tag{3}$$

5.4 Sample Configuration Code

The following code configures and starts the timer 1 module in a timer mode configuration:

```
1  void timer1_init(void){
2
3    T1CONbits.TCS   = 0; /* use internal clock: Fcy       */
4    T1CONbits.TGATE = 0; /* Gated mode off               */
5    T1CONbits.TCKPS = 1; /* prescale 1:8                 */
6    T1CONbits.TSIDL = 1; /* don't stop the timer in idle  */
7
8    TMR1 = 0;       /* clears the timer register            */
9    PR1 = M_SEC;      /* value at which the register overflows */
10
11   /* interruptions */
12   IPC0bits.T1IP = 5;   /* Timer 1 Interrupt Priority 0-7 */
13   IFS0bits.T1IF = 0;   /* clear interrupt flag           */
14   IEC0bits.T1IE = 1;   /* Timer 1 Interrupt Enable       */
15
16   T1CONbits.TON = 1;   /* starts the timer               */
17   return;
18 }
19
20
```

And the interruption code.

```
1    \*Timer1 Interruption*\
2      \*Some Code*\
3      IFS0bits.T1IF = 0;     /* clears interruption flag */
4    }
5
6
```

6 Output Compare

6.1 Introduction

Output Compare has a severe relevance due to is unique ability in the dsPIC30F6012A and dsPIC30F4013 to generate PWM signals. Its is the only way this dsPIC's have to generate PWM signals in an autonomous way. Again we will not cover all the working modes the module has to offer and we will only approach the PWM with no fault pin mode.

6.2 PWM Signals

PWM stands for Pulse-width Modulation, another similar acronym is PDM, Pulse-duration Modulation. As the name it's self states PWM is a modulation technique where we encode the amplitude of the signal into the width of the pulse (duration) of another signal.
This forces us to define the width of the pulse and we use the term/concept of *duty cycle* and we define it as the proportion between the time that the signal is at *HIGH* state and the time of the entire pulse. This signals are of major importance in motor control, system communication and for our specific case to create an analogical signal from a digital signal (DAC, Digital to Analog Conversion).

6.3 PWM mode

To create this PWM signal, we are going to use the output compare module, this module has a easy configuration setup. Its principle of working is based on a timer so you'll have to configure one timer that will support the OC module with a timing reference. You can use either Timer 2 or Timer 3, and you can configure this in *OCTSEL* bits in *OCxCON* Register.
The output signal will be generated on *OCx* Pin being x one of the eight available pins which are associated with the OC module.
The register in which we configure the module to perform a PWM signal is once again *OCxCON*, using the *OCM* bits. After performing the configuration of the Timer and the OC module, we can now control the signal using two dedicated registers:

- *OCxR*
- *OCxRS*

Now we are able to create the PWM signal. The first thing we have to understand is that the period of this signal equals the period of the timer we configured. Secondly the signal will start at *HIGH* state and will go to *LOW* state once the *TMRy* Register (being *y* the number of the timer you selected) matches the value on the *OCxR*. In third place we are not allowed to change the value on *OCxR* Register[4] so in order to change it, the value on the *OCxRS* Register will update to *OCxR* Register every period (meaning when *TMRy* value equals *PRy* value). Easy enough the only thing we must do to generate our signal is control its period with *PRy* value and the time at which the signal changes its state with *OCxRS* value.

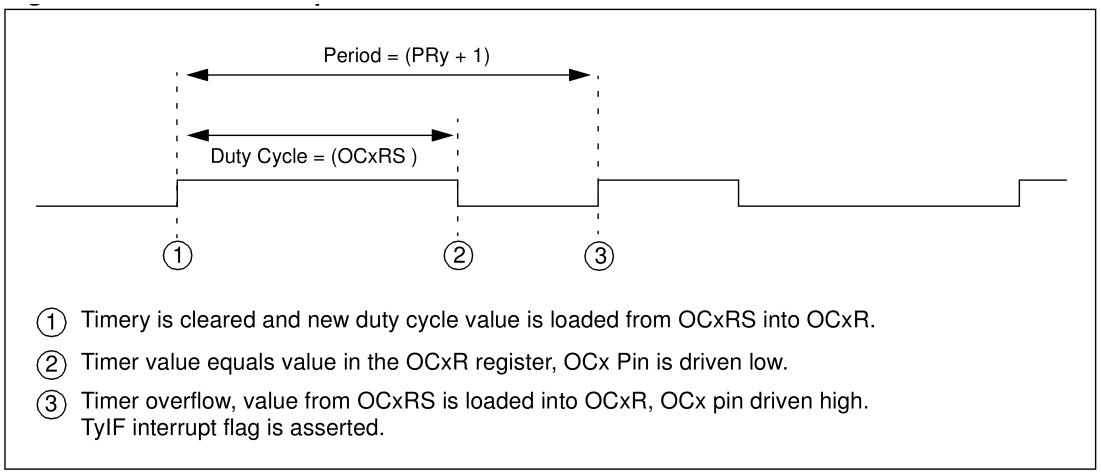

Figure 2: PWM Output Waveform

As we can see from this picture the PWM signal is controlled using the period and Duty Cycle.

[4]We are allowed but is not a good idea, so we will stay with the forbidden condition

6.4 Sample Configuration Code

The following code implements a standard configuration for PWM signal generation.

```
1   void PWM_setup()  /* Sec. 14.2 dsPIC */
2   {
3
4     /*Fosc = 30 MHz
5     * Tcy = 1/Fcy = 133.3 ns
6     * Tpwm = [(PR2) + 1] * Tcy * (Timer 2 Prescale Value) = 10.2 us
7     * Fpwm = 1/[Tpwm] ~= 98.04KHz
8     * Max Resolution = log10(Fosc/Fpwm) / log10(2) ~= 27.6 bits
9     */
10    OC1R = 0;      /* Init Duty Cicle                        */
11
12    OC1CONbits.OCSIDL = 0;     /* Stop on Idle Disable          */
13    OC1CONbits.OCTSEL = 0;     /* Select Timer 2 as Time Base   */
14    OC1CONbits.OCM    = 6;     /* Mode: PWM with no Fault Bit   */
15
16    return;
17  }
18
19
20  void PWM(int DutyCycle){
21    OC1RS = DutyCycle;
22  }
23
```

7 Analog to Digital Conversion

7.1 Introduction

The Analog to Digital Conversion module is probably the most extensive module in the dsPIC in what come to portability and configuration modes. It supports several manners of configuration, once again for simplicity we will only approach two ways of configuring the module.
Before entering in a detailed description about dsPIC module an general overview of ADC might be worthwhile.
As defined in wikipedia "An analog-to-digital converter (ADC, A/D, or A to D) is a device that converts a continuous physical quantity (usually voltage) to a digital number that represents the quantity's amplitude". This conversion therefore converts continuous quantity into a discrete quantity, so obviously an error is associated with the conversion, and furthermore, also as consequence of the discretization, a bandwidth is associated.

7.1.1 Resolution

Resolution is an important part of and ADC conversion and we should always be aware of it and describe it in our code. The Resolution describes the magnitude of the quantized error. The values will be stored in binary form, so resolution is expressed, as well as the value of the conversion, in bits. As a direct consequence the number of "levels" will be a power of two. dsPIC provides a 12-bit ADC so:

$$Resolution = 2^{12} = 4096 \tag{4}$$

This means we have 4096 levels at our disposal.
For a good quantization error analysis it is a good practice to look at the $SQNR$, Signal-to-quantization-noise ratio. There are several ways of calculating it depending your application. There are several ways of improving the resolution, like introducing white-noise, oversampling and conversion enhancing, none of which we will explore in this document.

7.1.2 Undersampling and Aliasing

Undersampling is a risk when applying a discretization and therefore we should always guarantee at least twice the frequency of sampling compared to the maximum frequency of the signal we are acquiring. Nyquist theorem states that if the above condition is not ensured we might enter in an Aliasing situation, and as a consequence we can not

fully reconstruct the signal that we are sampling. So it is also very important to have knowledge of the input signal bandwidth, and ensure a frequency high enough not to undersample the signal. If we really (really really) know what we are doing aliasing might come in hand to apply bandpass sampling, we are not going to perform any type of analysis on this subject but keep on mind that intentional undersampling is a powerful tool.
dsPIC provides a successive-approximation ADC and non-linearities may result from accumulating errors in the subtraction process. Channel smoothing is a way not discussed here to decrease the subtraction errors.

7.1.3 Speed and Precision

The speed of any ADC module is limited by their clock rate and conversion time. Clock rate is an outside characteristic and normally not in our hands, which leaves conversion time. Conversion time is a mixed blessing, diminishing it increases the ADC speed, but decreases the precision. When discussing precision linearity is an important subject. The ADC module in the dsPIC is linear, which means steeping from one level to the following the increase always correspond to the same amount of magnitude increase in the input signal. Not all ADC's in the market are linear, logarithmic ADC is also very common. Still virtually every application we will use in this project linear ADC is the way to go. Why is linearity an important subject when discussing precision and speed? We can evaluate our precision based on our linearity and low precisions (or high speeds) may result in pour linearities. Thermal effects have a considerable effect in the linearity and if possible should be taken into consideration. When dealing with analog sensors integer linearity is a useful tool to get a quantization of the sensor fidelity and therefore use it when complex active filters (e.g. Kalman Filter) are used.

7.2 Modes of Operation

As stated before dsPIC provides several modes of operation and only two of those will be covered.

- **Manual Sample**
  When in manual mode the conversion trigger is under software control. We can control this by setting or clearing the *SAMP* bit in *ADCON1*. Setting this bit will start sampling, and clearing this bit will stop sampling and start conversion.

- **Auto Sample**
  When in Auto Sample mode, **Clocked Conversion Trigger** is implied, this means

that the number of cycles between Sampling and Converting is defined in relation to T_{AD}. The time between sampling start and conversion start is defined as the sampling time, then,

$$T_{SMP} = SAMPC < 4:0 > \times T_{AD} \tag{5}$$

where *SAMC* bits are present in *ADCON3*.
Besides the conversion time being automatically defined by the *SAMP* bits, when *ASAM* in *ADCON1* bit is set then a new sample will start immediately after the conversion ended. This way the dspIC will be constantly converting and the whole process time will be,

$$ADCPeriod = SAMPC < 4:0 > \times T_{AD} + ConversionTime \tag{6}$$

The conversion time, despite the mode of operation, always is,

$$ConversionTime = 14 \times T_{AD} \tag{7}$$

7.3 Control Registers Overview

The Registers associated with the control of the ADC Module are:

- **ADCON1: A/D Control Register 1**
  In this register we activate the module, select the data output format, select the conversion trigger, configure auto sampling, set or clear the *SAMP* bit, and check if the conversion has been completed.

- **ADCON2: A/D Control Register 2**
  In this register we enable/disable scan mode, check the buffer filling status, configure the conversions per interrupt, select the buffer mode and the MUX to be used.

- **ADCON3: A/D Control Register 3**
  In this register we select Auto Sample time, Conversion clock source and select T_{AD} in function of T_{cy}

- **ADCHS: A/D Input Channel Select Register**
  In this register we select the negative and positive input for MUX A/B

- **ADPCFG: A/D Port Configuration Register**
  In this register we configure the pins from the analog port as either digital or analog

- **ADCSSL: A/D Input Scan Selection Register**
  In this Register we select the input pins for scan

7.4 Conversion Sequence

"A sampling of the analog input pin voltage is performed by sample and hold S/H amplifiers. The S/H amplifiers are also called S/H channels." When this sample ends the A/D is disconnected from the pin and conversion starts, taking a total time of $14T_{AD}$. When the conversion is complete the result is loaded into one of the 16 result registers (*ADCBUF0,..,ADCBUFF*). To ensure a good conversion you must make sure that T_{AD} value is grater than 333.33 nano-seconds.

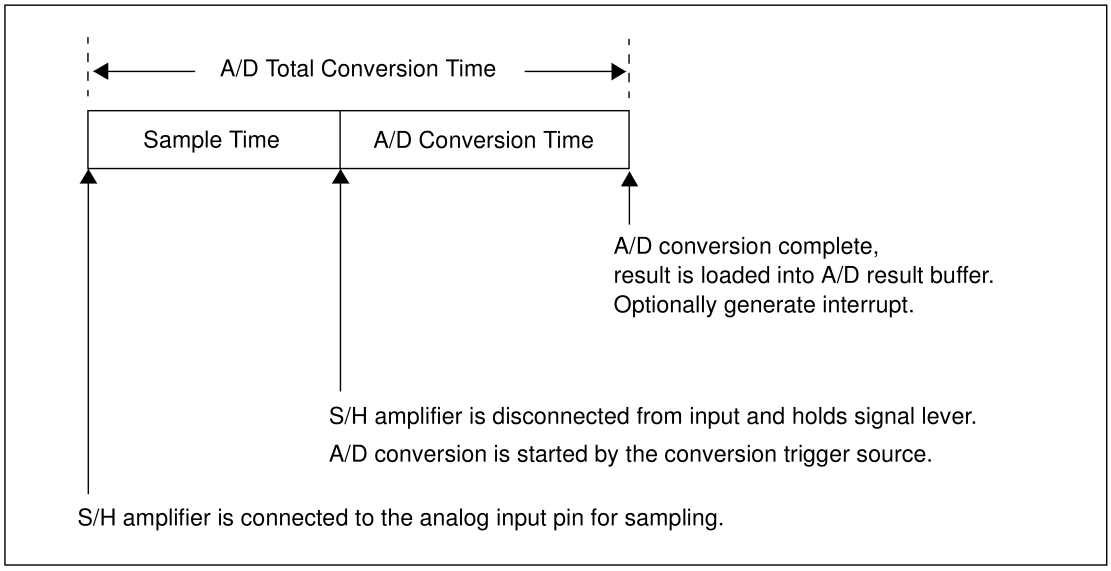

Figure 3: A/D Sample/Conversion Sequence

In the figure above we can see a typical Sample/Conversion sequence.

7.4.1 Conversion Clock

To select the Conversion clock we must attend the following equations:

$$T_{AD} = \frac{T_{CY}(ADCS + 1)}{2} \tag{8}$$

$$ADCS = \frac{2T_{AD}}{T_{CY}} - 1 \tag{9}$$

7.5 Module Configuration

The following steps should be followed for performing an A/D conversion:[5]

- **Configure the A/D module**
  - Select voltage reference source to match expected range on analog inputs
  - Select the analog conversion clock to match desired data rate with processor clock
  - Determine how sampling will occur
  - Determine how inputs will be allocated to the S/H channel
  - Select how conversion results are presented in the buffer
  - Select interrupt rate
  - Turn on A/D module
- **Configure A/D interrupt (if required)**
  - Clear ADIF bit
  - Select A/D interrupt priority

7.6 Buffer Filling

ADC module will fill the buffer in different ways concerning the mode of operation. In a single manual conversion like the one approached in the first code presented in the section **Sample Code** the result of the conversion will be placed in the first of the sixteen buffers provided (*ADCBUF0*).
In the auto sample mode with input scan, the ADC module will fill the number of buffers in function of the *SMPI* bits in *ADCON2*. For example if $SMPI = 4$, like in the second code provided at section **Sample Code**, the the first 4 buffers will be occupied, (*ADCBUF0*,...,*ADCBUF3*). It is a good practice to immediately retrieve the values of the buffers to a in code variable, and avoid using the buffer registers in operations.

[5] this section is identical to section 18.5 of the dsPIC Family Reference Manual

7.7 Sample Code

The following code implements a manual sample configuration.

```
1
2     void init_ADC12()
3     {
4
5     /*----ADCON 1 Configure----*/
6       ADCON1bits.ADSIDL = 0;  // Discontinue in IDLE
7       ADCON1bits.FORM   = 0;  // Integer output
8       ADCON1bits.SSRC   = 0;  // Clearing SAMP bit ends sampling and
    starts conversion
9       ADCON1bits.ASAM   = 0;  // Sampler auto-start
10    /*----ADCON2  Configure----*/
11      ADCON2bits.VCFG   = 0;  // Internal AVDD and AVSS
12      ADCON2bits.CSCNA  = 0;  // scan disable
13      ADCON2bits.SMPI   = 0;  // One sample per interrupt
14      ADCON2bits.BUFM   = 0;  // One 16 words buffer
15      ADCON2bits.ALTS   = 0;  // alternate MuxA and MuxB disable,
    Always use MUXA
16    /*----ADCON3  Configure----*/
17      ADCON3bits.SAMC   = 1;  // Sample time: 1 Tad
18      ADCON3bits.ADRC   = 0;  // Use system clock conversion
19      ADCON3bits.ADCS   = 21; // Conversion clock as T_CY*21
20    /*----ADCSSL----*/
21      ADCHS = 0;
22      ADCSSLbits.CSSL1 = 1; //ports 1 in conversion
23      ADPCFG  = 0;       //All pins analog
24    /*----Interruption Configure----*/
25      IPC2bits.ADIP   = 6;  //Interrup priority is 6
26      IEC0bits.ADIE   = 0;  //Interrup is disabled
27      IFS0bits.ADIF = 0;  //Clear Interrupt Flag
28    /*----Turn On----*/
29      ADCON1bits.ADON = 1;  // ADC on
30    }
31
32    unsigned int getADCvalue(){
33      unsigned int ADCValue;
34      ADCON1bits.SAMP = 1; // start sampling ...
35      __delay_ms(10);       // Sample Time
36      ADCON1bits.SAMP = 0; // start Converting
37      while (!ADCON1bits.DONE); // conversion done?
38      ADCValue = ADCBUF0; // yes then get ADC value
39      return (ADCValue);  // Return ADC value
40    }
41
42
```

The following code implements a Auto start Configuration

```
void init_ADC12()
{

  TRISBbits.TRISB2  = 1;
  TRISBbits.TRISB3  = 1;
  TRISBbits.TRISB4  = 1;
  TRISBbits.TRISB5  = 1;
/*-----ADCON 1 Configure-----*/
  ADCON1bits.ADSIDL = 0;     // Discontinue in IDLE
  ADCON1bits.FORM   = 0;     // Integer output
  ADCON1bits.SSRC   = 7;     // Auto convert
  ADCON1bits.ASAM   = 1;     // Sampler auto-start
/*-----ADCON  Configure-----*/
  ADCON2bits.VCFG   = 0;  // Internal AVDD and AVSS
  ADCON2bits.CSCNA  = 1;  // scan enable
  ADCON2bits.SMPI   = 4;  // 4 samples per interrupt
  ADCON2bits.BUFM   = 0;  // One 16 words buffer
  ADCON2bits.ALTS   = 0;  // alternate MuxA and MuxB disable, Always
   use MUXA
/*-----ADCON  Configure-----*/
  ADCON3bits.SAMC   = 1;  // Auto sample time: 1 Tad
  ADCON3bits.ADRC   = 0;  // Use system clock conversion
  ADCON3bits.ADCS   = 21; // Conversion clock as T_CY*21
/*-----ADCSSL-----*/
  ADCHS = 0;
  ADCSSL  = 0x003C; //ports 2,3,4,5 in conversion
  ADPCFG  = 0;
/*-----Interruption Configure-----*/
  IPC2bits.ADIP   = 6;  //Interrup priority is 6
  IEC0bits.ADIE   = 1;  //Interrup is disabled
  IFS0bits.ADIF = 0;  //Clear Interrupt Flag
/*-----Turn On-----*/
  ADCON1bits.ADON=1;     // ADC on
}


/*--------------ADC Interruption-----------------*/
void __attribute__((interrupt, no_auto_psv)) _ADCInterrupt(void){

  Value[0] = ADCBUF0;
  Value[1] = ADCBUF1;
  Value[2] = ADCBUF2;
  Value[3] = ADCBUF3;

  IFS0bits.ADIF=0;
  return;
}


```