CHAPTER 10

Analog Input

The PIC32 has one analog-to-digital converter (ADC) that, through the use of multiplexers, can sample the analog voltage from 16 pins (Port B). Typically used with sensors that produce analog voltages, the ADC can capture nearly one million readings per second. The ADC has 10-bit resolution, which means it distinguishes $2^{10} = 1024$ voltage values, usually in the range from 0 to 3.3 V, yielding approximately 3 mV resolution. For higher resolution analog inputs, you can use an external chip and communicate with it using SPI (Chapter 12) or I^2C (Chapter 13).

10.1 Overview

Analog to digital conversion is a multi-step process. First the voltage on the appropriate pin must be routed to an internal differencing amplifier, which outputs the difference between the pin voltage and a reference voltage. Next, the voltage difference is sampled and held by an internal capacitor. Finally, the ADC converts the voltage on the capacitor into a 10-bit binary number.

Figure 10.1 shows a block diagram of the ADC, adapted from the Reference Manual. First we must determine which signals feed the differencing amp, which is located near the middle of Figure 10.1. Control logic (determined by SFRs) selects the differencing amp's + input from the analog pins AN0 to AN15 and the − input from either AN1 or V_{REFL}, a selectable reference voltage.[1] For proper operation, the − input voltage V_{INL} should be less than or equal to the + input voltage V_{INH}.

The differencing amp sends the difference of the two input voltages, $V_{SHA} = V_{INH} - V_{INL}$, to the Sample and Hold Amplifier (SHA). During the *sampling* (or *acquisition*) stage, a 4.4 pF internal holding capacitor charges or discharges to hold the voltage difference V_{SHA}. Once the sampling period has ended, the SHA is disconnected from the inputs, allowing V_{SHA} to remain constant during the *conversion* stage, even if the input voltages change.

[1] This reference V_{REFL} can be chosen to be either V_{REF-}, a voltage provided on an external pin, or AV_{SS}, the PIC32's GND line, also known as V_{SS}.

Embedded Computing and Mechatronics with the PIC32 Microcontroller. http://dx.doi.org/10.1016/B978-0-12-420165-1.00010-X

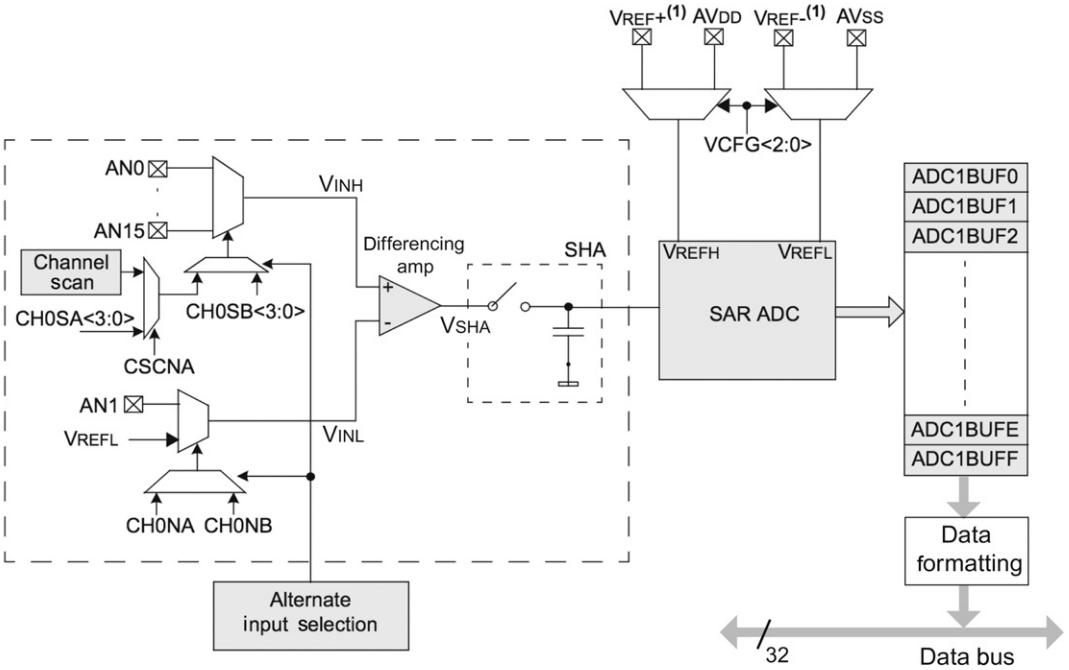

Figure 10.1
A simplified schematic of the ADC module.

The Successive Approximation Register (SAR) converts V_{SHA} to a 10-bit result depending on the low (V_{REFL}) and high (V_{REFH}) reference voltages: $1024 \times V_{SHA}/(V_{REFH} - V_{REFL})$, rounded to the nearest integer between 0 and 1023. (See the Reference Manual for more details on the ADC transfer function.) The 10-bit conversion result is written to the buffer ADC1BUF which is read by your program. If you do not read the result right away, ADC1BUF can store up to 16 results (in the SFRs ADC1BUF0, ADC1BUF1, …, ADC1BUFF) before the ADC begins overwriting old results.[2]

Sampling and conversion timing

The two main stages of an ADC read are sampling/acquisition and conversion. During the sampling stage, we must allow sufficient time for the internal holding capacitor to converge to the difference $V_{INH} - V_{INL}$. According to the Electrical Characteristics section of the Data Sheet, this time is 132 ns when the SAR ADC uses the external voltage references V_{REF-} and

[2] The ADC1BUFx buffers are not contiguous in memory. Each buffer is four bytes long, but they are 16 bytes apart.

V_{REF+} as its low and high references. The minimum sampling time is 200 ns when using AV_{SS} and AV_{DD} as the low and high references.

Once the sampling stage finishes, the SAR begins the conversion process, using successive approximation to find the digital representation of the voltage. This method uses a binary search, iteratively comparing V_{SHA} to test voltages produced by an internal digital-to-analog converter (DAC). The DAC converts 10-bit numbers into test voltages between V_{REFL} and V_{REFH}: 0x000 produces V_{REFL} and 0x3FF produces V_{REFH}. During the first iteration, the DAC's test value is 0x200 = 0b1000000000, which produces a voltage in the middle of the reference voltage range. If V_{SHA} is greater than this DAC voltage, the first result bit is one, otherwise it is zero. On the second cycle, the DAC's most significant bit is set to the first test's result and the second most significant bit is set to 1. The comparison is performed and the second result bit determined. The process continues until all 10 bits of the result are determined. The entire process requires 10 cycles, plus 2 more, for a total of 12 ADC clock cycles.

The ADC clock is derived from PBCLK. According to the Electrical Characteristics section of the Data Sheet, the ADC clock period (Tad) must be at least 65 ns to allow enough time to convert a single bit. The ADC SFR AD1CON3 allows us to choose the ADC clock period as $2 \times k \times Tpb$, where Tpb is the PBCLK period and k is any integer from 1 to 256. Since Tpb is 12.5 ns for the NU32, to meet the 65 ns specification, the smallest value we can choose is k = 3, or Tad = 75 ns.

The minimum time between samples is the sum of the sampling time and the conversion time. If configured to sample automatically, we must choose the sampling time to be an integer multiple of Tad. The shortest time we can choose is $2 \times$ Tad = 150 ns to satisfy the 132 ns minimum sampling time. Thus the fastest we can read from an analog input is

```
minimum read time = 150 ns + 12 * 75 ns = 1050 ns
```

or just over 1 μs. We can, theoretically, read the ADC at almost one million samples per second (1 MHz).

Multiplexers

Two multiplexers determine which analog input pins to connect to the differencing amp. These two multiplexers are called MUX A and MUX B. MUX A is the default active multiplexer, and the SFR AD1CON3 contains CH0SA bits that determine which of AN0 to AN15 is connected to the + input and CH0NA bits that determine which of AN1 and V_{REF-} is connected to the − input. It is possible to alternate between MUX A and MUX B, but you are unlikely to need this feature.

Options

The ADC peripheral provides a bewildering array of options, some of which are described here. No need to remember them all! The sample code provides a good starting point.

- Data format: The result of the conversion is stored in a 32-bit word, and it can be represented as a signed integer, unsigned integer, fractional value, etc. Typically we use either 16-bit or 32-bit unsigned integers.
- Sampling and conversion initiation events: Sampling can be initiated by a software command or immediately after the previous conversion has completed (auto sample). Conversion can be initiated by a software command, the expiration of a specified sampling period (auto convert), a period match with Timer3, or a signal change on the INT0 pin. If sampling and conversion happen automatically (i.e., not through software commands), the conversion results are placed in the ADC1BUF at successively higher addresses (ADC1BUF0 to ADC1BUFF) before returning to the first address in ADC1BUF.
- Input scan and alternating modes: You can read one analog input at a time, scan through a list of inputs (using MUX A), or alternate between two inputs (one from MUX A and one from MUX B).
- Voltage reference: The ADC normally uses reference voltages of 0 and 3.3 V (the power rails of the PIC32); therefore, a reading of 0x000 corresponds to 0 V and a reading of 0x3FF corresponds to 3.3 V. If you are interested in a different voltage range—say 1.0 V to 2.0 V—you can configure the ADC so that 0x000 corresponds to 1.0 V and 0x3FF corresponds to 2.0 V, giving you better resolution: $(2\text{ V} - 1\text{ V})/1024 = 1\text{ mV}$ resolution. You supply alternate voltage references on pins V_{REF-} and V_{REF+}. The voltages provided must be between 0 and 3.3 V.
- Unipolar differential mode: Any of the analog inputs ANx (x = 2 to 15, e.g., AN5) can be compared to AN1, allowing you to read the voltage difference between ANx and AN1. The voltage on ANx should be greater than the voltage on AN1.
- Interrupts: An interrupt may be generated after a specified number of conversions. The number of conversions per interrupt also determines which ADC1BUFx buffer is used, even if you do not enable the interrupt. Conversion results are placed in successively higher numbered ADC1BUFx buffers (i.e., the first conversion goes in ADC1BUF0, the next in ADC1BUF1, etc.). When the interrupt triggers, the current buffer wraps around to ADC1BUF0 (or, in dual buffer mode, ADC1BUF8, see below). So if you set the ADC to interrupt on every conversion (the default), the results will always be stored in ADC1BUF0.
- ADC clock period: The ADC clock period Tad can range from 2 times the PB clock period up to 512 times the PB clock period, in integer multiples of two. Tad must be long enough to convert a single bit (65 ns according to the Electrical Characteristics section of the Data Sheet). You may also choose Tad to be the period of the ADC internal RC clock.

- Dual buffer mode: When an ADC conversion finishes, the result is written into the output buffer ADC1BUFx (x = 0x0 to 0xF). The ADC can be configured to write a series of conversions into a sequence of output buffers. The first conversion is stored in ADC1BUF0, the second in ADC1BUF1, etc. After a series of conversions, an interrupt flag is set, indicating that the results are available for the program to read. The next set of conversions starts over at ADC1BUF0; if the program is slow to read the results, the next conversions may overwrite the previous results. To help with this scenario, the 16 ADC1BUFx buffers can be split into two 8-word groups: one in which the current conversions are written, and one from which the program should read the results. The first conversion sequence starts writing at ADC1BUF0, the next starts at ADC1BUF8, and the starting buffers alternate from there.

10.2 Details

The operation of the ADC peripheral is determined by the following SFRs, all of which default to all zeros on reset.

AD1PCFG Only the least significant 16 bits are relevant. If a bit is 0, the associated pin on port B is configured as an analog input. If a bit is 1, it is digital I/O. The analog input pins AN0 to AN15 correspond to the port B pins RB0 to RB15.

AD1CON1 One of three main ADC control registers: controls the output format and conversion and sampling methods.

AD1CON1⟨15⟩ or AD1CON1bits.ON: Enables and disables the ADC.

1 The ADC is enabled.

0 The ADC is disabled.

AD1CON1⟨10:8⟩ or AD1CON1bits.FORM: Determines the data output format. We usually use either

0b100 32-bit unsigned integer

0b000 16-bit unsigned integer (the default).

AD1CON1⟨7:5⟩ or AD1CON1bits.SSRC: Determines what begins the conversion process. The two most common methods are

0b111 Auto conversion. The conversion begins as soon as sampling ends. Hardware automatically clears AD1CON1bits.SAMP to begin the conversion.

0b000 Manual conversion. You must clear AD1CONbits.SAMP to start the conversion.

AD1CON1⟨2⟩ or AD1CON1bits.ASAM: Determines whether another sample occurs immediately after conversion.

1 Use auto sampling. Sampling starts after the last conversion is finished. Hardware automatically sets AD1CON1bits.SAMP.

0 Use manual sampling. Sampling begins when the user sets AD1CON1bits.SAMP.

AD1CON1⟨1⟩ or AD1CON1bits.SAMP: Indicates whether the sample and hold amplifier (SHA) is sampling or holding. When auto sampling is disabled (AD1CON1bits.ASAM = 0), set this bit to initiate sampling. When using manual conversion (AD1CON1bits.SSRC = 0) clear this bit to zero to start conversion.

1 The SHA is sampling. Setting this bit initiates sampling when in manual sampling mode (AD1CON1bits.ASAM = 0).

0 The SHA is holding. Clearing this bit begins conversion when in manual conversion mode (AD1CON1bits.SSRC = 0).

AD1CON1⟨0⟩ or AD1CON1bits.DONE: Indicates whether a conversion is occurring. When using automatic sampling, hardware clears this bit automatically.

1 The analog-to-digital conversion is finished.

0 The analog-to-digital conversion is either pending or has not begun.

AD1CON2 Determines voltage reference sources, input pin selections, and the number of conversions per interrupt.

AD1CON2⟨15:13⟩ or AD1CON2bits.VCFG: Determines the voltage reference sources for the V_{REFH} and V_{REFL} inputs to the SAR. These references determine what voltage a given reading corresponds to: 0x000 corresponds to V_{REFL} and 0x3FF corresponds to V_{REFH}.

0b000 Use the internal references: V_{REFH} is 3.3 V and V_{REFL} is 0 V.

0b001 Use an external reference for V_{REFH} and an internal reference for V_{REFL}: V_{REFH} is the voltage on the V_{REF+} pin and V_{REFL} is 0 V.

0b010 Use an internal reference for V_{REFH} and an external reference for V_{REFL}: V_{REFH} is 3.3 V and V_{REFL} is the voltage on the V_{REF-} pin.

0b011 Use external references: V_{REFH} is the voltage on the V_{REF+} pin and V_{REFL} is the voltage on the V_{REF-} pin.

AD1CON2⟨10⟩ or AD1CON2bits.CSNA: Control scanning of inputs. The pins to scan are selected by AD1CSSL.

1 Scan inputs. Each subsequent sample will be from a different pin, selected by AD1CSSL, wrapping around to the beginning when the last pin is reached.

0 Do not scan inputs. Only one input is used.

AD1CON2⟨7⟩ or AD1CON2bits.BUFS: Used only in split buffer mode (AD1CON2bits.BUFM = 1). Indicates which buffer the ADC is currently filling.

1 The ADC is filling buffers ADC1BUF8 to ADC1BUFF, so the user should read from buffers ADC1BUF0 to ADC1BUF7.

0 The ADC is filling buffers ADC1BUF0 to ADC1BUF7, so the user should read from buffers ADC1BUF8 to ADC1BUFF.

AD1CON2⟨5:2⟩ or AD1CON2bits.SMPI: The number of sample/conversion sequences per interrupt is AD1CON2bits.SMPI + 1. In addition to determining when the interrupt occurs, these bits also determine how many conversions must occur before the ADC starts storing data in the first buffer (or the alternate first buffer when AD1CON2bits.BUFM = 1). For example, if AD1CON2bits.SMPI = 1 then there will

be two conversions per interrupt. The first conversion will be stored in AD1BUF0 and the second in AD1BUF1. After the second conversion the ADC interrupt flag will be set. The next conversion will be stored in AD1BUF0 if AD1CON2bits.BUFM = 0 or AD1BUF8 if AD1CON2bits.BUFM = 1.

AD1CON2⟨1⟩ or AD1CON2bits.BUFM: Determines if the ADC buffer is split into two 8-word buffers or is used as a single 16-word buffer.

1 The ADC buffer is split into two 8-word buffers, ADC1BUF0 to ADC1BUF7 and ADC1BUF8 to ADC1BUFF. Data is alternatively stored in the lower and upper buffers, every AD1CON2bits.SMPI + 1 sample/conversion sequences.

0 The ADC buffer is used as a single 16 word buffer, ADC1BUF0 to ADC1BUFF.

AD1CON3 Controls settings for the ADC clock and other ADC timing settings. Determines Tad, the ADC clock period.

AD1CON3⟨12:8⟩ or AD1CON3bits.SAMC: Determines the length of auto sampling time, in Tad. Can be set anywhere from 1 Tad to 31 Tad; however, sampling requires at least 132 ns.

AD1CON3⟨7:0⟩ or AD1CON3bits.ADCS: Determines the length of Tad, in terms of the peripheral bus clock period Tpb, according to the formula

$$\text{Tad} = 2 \times \text{Tpb} \times (\text{AD1CON3bits.ADCS} + 1). \tag{10.1}$$

Tad must be at least 65 ns, so with an 80 MHz peripheral bus clock frequency, the minimum value for AD1CON3bits.ADCS is 2, which yields a 75 ns Tad by the equation above.

AD1CHS This SFR determines which pins will be sampled (the "positive" inputs) and what they will be compared to (i.e., V_{REFL} or AN1). When in scan mode, the sample pins specified in this SFR are ignored. There are two multiplexers available, MUX A and MUX B; we focus on the settings for MUX A.

AD1CHS⟨23⟩ or AD1CHSbits.CH0NA: Determines the negative input for MUX A. When MUX A is selected (the default), this input is the negative input to the differencing amplifier.

1 The negative input is the pin AN1.

0 The negative input is V_{REFL}. V_{REFL} is determined by AD1CON2bits.VCFG.

AD1CHS⟨19:16⟩ or AD1CHSbits.CH0SA: Determines the positive input to MUX A. When MUX A is selected (the default), this input is the positive input to the differencing amplifier. The value of this field determines which ANx pin is used. For example, if AD1CHS.CHOSA = 6 than AN6 is used.

AD1CSSL Bits set to 1 in this SFR indicate which analog inputs will be sampled in scan mode (if AD1CON2 has configured the ADC for scan mode). Inputs will be scanned from lower number inputs to higher numbers. Bit x corresponds to an ANx. Individual bits can be accessed using AD1CSSLbits.CSSLx.

Apart from these SFRs, the ADC module has bits associated with the ADC interrupt in IFS1bits.AD1IF (IFS1⟨1⟩), IEC1bits.AD1IE (IEC1⟨1⟩), IPC6bits.AD1IP (IPC6⟨28:26⟩), and IPC6bits.AD1IS (IPC6⟨25:24⟩). The interrupt vector is 27, also known as `_ADC_VECTOR`.

10.3 Sample Code

10.3.1 Manual Sampling and Conversion

There are many ways to read the analog inputs, but the sample code below is perhaps the simplest. This code reads in analog inputs AN14 and AN15 every half second and sends their values to the user's terminal. It also logs the time it takes to do the two samples and conversions, which is a bit under 5 μs total. In this program we set the ADC clock period Tad to be $6 \times \text{Tpb} = 75$ ns, and the acquisition time to be at least 250 ns. There are two places in this program where we wait and do nothing: during the sampling and during the conversion. If speed were an issue, we could use more advanced settings to let the ADC work in the background and interrupt when samples are ready.

In the exercises you will write code to initiate the conversion automatically, rather than manually as in the sample code below.

Code Sample 10.1 `ADC_Read2.c`. **Reading Two Analog Inputs with Manual Initialization of Sampling and Conversion.**

```
#include "NU32.h"          // constants, functions for startup and UART

#define VOLTS_PER_COUNT (3.3/1024)
#define CORE_TICK_TIME 25    // nanoseconds between core ticks
#define SAMPLE_TIME 10       // 10 core timer ticks = 250 ns
#define DELAY_TICKS 20000000 // delay 1/2 sec, 20 M core ticks, between messages

unsigned int adc_sample_convert(int pin) { // sample & convert the value on the given
                                           // adc pin the pin should be configured as an
                                           // analog input in AD1PCFG
    unsigned int elapsed = 0, finish_time = 0;
    AD1CHSbits.CH0SA = pin;                // connect chosen pin to MUXA for sampling
    AD1CON1bits.SAMP = 1;                  // start sampling
    elapsed = _CP0_GET_COUNT();
    finish_time = elapsed + SAMPLE_TIME;
    while (_CP0_GET_COUNT() < finish_time) {
      ;                                    // sample for more than 250 ns
    }
    AD1CON1bits.SAMP = 0;                  // stop sampling and start converting
    while (!AD1CON1bits.DONE) {
      ;                                    // wait for the conversion process to finish
    }
    return ADC1BUF0;                       // read the buffer with the result
}

int main(void) {
  unsigned int sample14 = 0, sample15 = 0, elapsed = 0;
```

```
  char msg[100] = {};

  NU32_Startup();                 // cache on, interrupts on, LED/button init, UART init
  AD1PCFGbits.PCFG14 = 0;                 // AN14 is an adc pin
  AD1PCFGbits.PCFG15 = 0;                 // AN15 is an adc pin
  AD1CON3bits.ADCS = 2;                   // ADC clock period is Tad = 2*(ADCS+1)*Tpb =
                                          //                           2*3*12.5ns = 75ns
  AD1CON1bits.ADON = 1;                   // turn on A/D converter
  while (1) {
    _CP0_SET_COUNT(0);                    // set the core timer count to zero
    sample14 = adc_sample_convert(14);    // sample and convert pin 14
    sample15 = adc_sample_convert(15);    // sample and convert pin 15
    elapsed = _CP0_GET_COUNT();           // how long it took to do two samples
                                          // send the results over serial
    sprintf(msg, "Time elapsed: %5u ns  AN14: %4u (%5.3f volts)"
                 "  AN15: %4u (%5.3f volts) \r\n",
                 elapsed * CORE_TICK_TIME,
                 sample14, sample14 * VOLTS_PER_COUNT,
                 sample15, sample15 * VOLTS_PER_COUNT);
    NU32_WriteUART3(msg);
    _CP0_SET_COUNT(0);                    // delay to prevent a flood of messages
    while(_CP0_GET_COUNT() < DELAY_TICKS) {
      ;
    }
  }
  return 0;
}
```

If AN14 is connected to 0 V and AN15 is connected to 3.3 V, typical output of the program repeats the following two lines,

```
  ...
Time elapsed:  4550 ns  AN14:    0 (0.000 volts)  AN15: 1023 (3.297 volts)
Time elapsed:  4675 ns  AN14:    0 (0.000 volts)  AN15: 1023 (3.297 volts)
  ...
```

indicating that the two conversions take less than 5 μs with some minor variation each time through the loop.

10.3.2 Maximum Possible Sample Rate

The program `ADC_max_rate.c` reads from a single analog input, AN2, at the maximum speed that fits the PIC32 Electrical Characteristics and the 80 MHz PBCLK (Tpb = 12.5 ns). We choose

```
Tad = 6 * Tpb = 75 ns
```

as the smallest time that is an even integer multiple of Tpb and greater than the 65 ns required in the Electrical Characteristics section of the Data Sheet. We choose the sample time to be

```
Tsamp = 2 * Tad = 150 ns,
```

the smallest integer multiple of Tad that meets the minimum spec of 132 ns in the Data Sheet.[3] The ADC is configured to auto-sample and auto-convert eight samples and then generate an interrupt. The ISR reads eight samples from ADCBUF0 to ADCBUF7 or from ADCBUF8 to ADCBUFF while the ADC fills the other eight-word section. The ISR must finish reading one eight-word section before the other eight-word section is filled. Otherwise, the ADC results will start overwriting unread results.

After reading 1000 samples, the ADC interrupt is disabled to free the CPU from servicing the ISR. The program writes the data and average sample/conversion times to the user's terminal.

High-speed sampling requires pin V_{REF-} (RB1) to be connected to ground and pin V_{REF+} (RB0) to be connected to 3.3 V. These pins are the external low and high voltage references for analog input. Technically, the Reference Manual states that V_{REF-} should be attached to ground through a 10 ohm resistor and V_{REF+} should be attached to two capacitors in parallel to ground (0.1 μF and 0.01 μF) as well as a 10 ohm resistor to 3.3 V.

To provide input to the ADC, we configure OC1 to output a 5 kHz 25% duty cycle square wave. The program also uses Timer45 to time the duration between ISR entries. The first 1000 analog input samples are written to the screen, as well as the time they were taken, confirming that the samples correspond to 889 kHz sampling of a 5 kHz 25% duty cycle waveform. The ISR that reads eight samples from ADCBUF also toggles an LED once every million times it is entered, allowing you to measure the time it takes to acquire eight million samples with a stopwatch (about 9 s).

Code Sample 10.2 `ADC_max_rate.c`. **Reading a Single Analog Input at the Maximum Possible Rate to Meet the Electrical Characteristics Section of the Data Sheet, Given That the PBCLK Is 80 MHz.**

```
// ADC_max_rate.c
//
// This program reads from a single analog input, AN2, at the maximum speed
// that fits the PIC32 Electrical Characteristics and the 80 MHz PBCLK
// (Tpb = 12.5 ns).  The input to AN2 is a 5 kHz 25% duty cycle PWM from
// OC1.  The results of 1000 analog input reads is sent to the user's
// terminal.  An LED on the NU32 also toggles every 8 million samples.
//
// RB1/VREF- must be connected to ground and RB0/VREF+ connected to 3.3 V.
//

#include "NU32.h"                   // constants, functions for startup and UART

#define NUM_ISRS 125                // the number of 8-sample ISR results to be printed
#define NUM_SAMPS (NUM_ISRS*8)      // the number of samples stored
#define LED_TOGGLE 1000000          // toggle the LED every 1M ISRs (8M samples)
```

[3] The Electrical Characteristics section of the Data Sheet lists 132 ns as the minimum sampling time for an analog input from a source with 500 Ω output impedance. If the source has a much lower output impedance, you may be able to reduce the sampling time below 132 ns.

```
// these variables are static because they are not needed outside this C file
// volatile because they are written to by ISR, read in main

static volatile int storing = 1;   // if 1, currently storing data to print; if 0, done
static volatile unsigned int trace[NUM_SAMPS];   // array of stored analog inputs
static volatile unsigned int isr_time[NUM_ISRS]; // time of ISRs from Timer45

void __ISR(_ADC_VECTOR, IPL6SRS) ADCHandler(void) { // interrupt every 8 samples
  static unsigned int isr_counter = 0; // the number of times the isr has been called
                                       // "static" means the variable maintains its value
                                       // in between function (ISR) calls
  static unsigned int sample_num = 0;  // current analog input sample number

  if (isr_counter <= NUM_ISRS) {
    isr_time[isr_counter] = TMR4;      // keep track of Timer45 time the ISR is entered
  }

  if (AD1CON2bits.BUFS) {              // 1=ADC filling BUF8-BUFF, 0=filling BUF0-BUF7
    trace[sample_num++] = ADC1BUF0;    // all ADC samples must be read in, even
    trace[sample_num++] = ADC1BUF1;    // if we don't want to store them, so that
    trace[sample_num++] = ADC1BUF2;    // the interrupt can be cleared
    trace[sample_num++] = ADC1BUF3;
    trace[sample_num++] = ADC1BUF4;
    trace[sample_num++] = ADC1BUF5;
    trace[sample_num++] = ADC1BUF6;
    trace[sample_num++] = ADC1BUF7;
  }
  else {
    trace[sample_num++] = ADC1BUF8;
    trace[sample_num++] = ADC1BUF9;
    trace[sample_num++] = ADC1BUFA;
    trace[sample_num++] = ADC1BUFB;
    trace[sample_num++] = ADC1BUFC;
    trace[sample_num++] = ADC1BUFD;
    trace[sample_num++] = ADC1BUFE;
    trace[sample_num++] = ADC1BUFF;
  }
  if (sample_num >= NUM_SAMPS) {
    storing = 0;                       // done storing data
    sample_num = 0;                    // reset sample number
  }
  ++isr_counter;                       // increment ISR count
  if (isr_counter == LED_TOGGLE) {     // toggle LED every 1M ISRs (8M samples)
    LATFINV = 0x02;
    isr_counter = 0;                   // reset ISR counter
  }

  IFS1bits.AD1IF = 0;                  // clear interrupt flag
}

int main(void) {
  int i = 0, j = 0, ind = 0;      // variables used for indexing
  float tot_time = 0.0;           // time between 8 samples
  char msg[100] ={};              // buffer for writing messages to uart
  unsigned int prev_time = 0;     // used for calculating time differences

  NU32_Startup(); // cache on, min flash wait, interrupts on, LED/button init, UART init

  __builtin_disable_interrupts(); // INT step 2: disable interrupts
```

```
                                // configure OC1 to use T2 to make 5 kHz 25% DC
  PR2 = 15999;                  // (15999+1)*12.5ns = 200us period = 5kHz
  T2CONbits.ON = 1;             // turn on Timer2
  OC1CONbits.OCM = 0b110;       // OC1 is PWM with fault pin disabled
  OC1R = 4000;                  // hi for 4000 counts, lo for rest (25% DC)
  OC1RS = 4000;
  OC1CONbits.ON = 1;            // turn on OC1

                                // set up Timer45 to count every pbclk cycle
  T4CONbits.T32 = 1;            // configure 32-bit mode
  PR4 = 0xFFFFFFFF;             // rollover at the maximum possible period, the default
  T4CONbits.TON = 1;            // turn on Timer45

                                // INT step 3: configure ADC generating interrupts
  AD1PCFGbits.PCFG2 = 0;        //        make RB2/AN2 an analog input (the default)
  AD1CHSbits.CHOSA = 2;         //        AN2 is the positive input to the sampler
  AD1CON3bits.SAMC = 2;         //        sample for 2 Tad
  AD1CON3bits.ADCS = 2;         //        Tad = 6*Tpb
  AD1CON2bits.VCFG = 3;         //        external Vref+ and Vref- for VREFH and VREFL
  AD1CON2bits.SMPI = 7;         //        interrupt after every 8th conversion
  AD1CON2bits.BUFM = 1;         //        adc buffer is two 8-word buffers
  AD1CON1bits.FORM = 0b100;     //        unsigned 32 bit integer output
  AD1CON1bits.ASAM = 1;         //        autosampling begins after conversion
  AD1CON1bits.SSRC = 0b111;     //        conversion starts when sampling ends
  AD1CON1bits.ON = 1;           //        turn on the ADC
  IPC6bits.AD1IP = 6;           // INT step 4: IPL6, to use shadow register set
  IFS1bits.AD1IF = 0;           // INT step 5: clear ADC interrupt flag
  IEC1bits.AD1IE = 1;           // INT step 6: enable ADC interrupt
  __builtin_enable_interrupts(); // INT step 7: enable interrupts at CPU

  TMR4 = 0;                     // start timer 4 from zero
  while(storing) {
    ;                           // wait until first NUM_SAMPS samples taken
  }
  IEC1bits.AD1IE = 0;           // disable ADC interrupt

  sprintf(msg,"Values of %d analog reads\r\n",NUM_SAMPS);
  NU32_WriteUART3(msg);
  NU32_WriteUART3("Sample #   Value   Voltage   Time");

  for (i = 0; i < NUM_ISRS; ++i) {// write out NUM_SAMPS analog samples
    for (j = 0; j < 8; ++j) {
      ind = i * 8 + j;          // compute the index of the current sample
      sprintf(msg,"\r\n%5d %10d %9.3f ", ind, trace[ind], trace[ind]*3.3/1024);
      NU32_WriteUART3(msg);
    }
    tot_time = (isr_time[i] - prev_time) *0.0125; // total time elapsed, in microseconds
    sprintf(msg,"%9.4f us; %d timer counts; %6.4f us/read for last 8 reads",
        tot_time, isr_time[i]-prev_time,tot_time/8.0);
    NU32_WriteUART3(msg);
    prev_time = isr_time[i];
  }

  NU32_WriteUART3("\r\n");
  IEC1bits.AD1IE = 1;         // enable ADC interrupt. won't print the information again,
                              // but you can see the light blinking
  while(1) {
    ;
  }
  return 0;
}
```

The output should look like

```
... (earlier output snipped)
928       1019     3.284
929       1015     3.271
930       1015     3.271
931       1015     3.271
932       1015     3.271
933          4     0.013
934          4     0.013
935          4     0.013    9.0000 us; 720 timer counts; 1.1250 us/read for last
8 reads  ... (later output snipped)
```

showing the sample number, the ADC counts, and the corresponding actual voltage for samples 0 to 999. The high output voltage from OC1 is measured as approximately 3.27 V, and the low output voltage is measured as approximately 0.01 V. You can also see that the output is high for 45 consecutive samples and low for 135 consecutive samples, corresponding to the 25% duty cycle of OC1. In the snippet above, OC1's switch from high to low is measured at sample 933.

Theoretically, the time needed for one sample is 150 ns + (12 × 75 ns) = 1050 ns, but we get an extra 1 Tad (75 ns or 6 Tpb) for 1125 ns. This is 888.89 kHz sampling. Where does the extra 75 ns come from? It's not due to the extra processing time needed to enter the interrupt and read the timer: these times are constant and cancel each other when measuring the time between two interrupts. Rather, the discrepancy is from the time needed to start conversion after sampling and the time needed to start sampling after conversion. The Electrical Characteristics section of the Data Sheet lists the "Conversion Start from Sample Trigger" as being typically 1.0 Tad and the "Conversion Completion to Sample Start" as being typically 0.5 Tad. Our experiment indicates that our measured times are actually a little lower than the listed typical values, since we have only 1 Tad, not 1.5 Tad, of unexpected sample/conversion time.

10.4 Chapter Summary

- The ADC peripheral converts an analog voltage to a 10-bit digital value, where 0x000 corresponds to an input voltage at V_{REFL} (typically GND) and 0x3FF corresponds to an input voltage at V_{REFH} (typically 3.3 V). There is a single ADC on the PIC32, AD1, but it can be multiplexed to sample from any or all of the 16 pins on Port B.
- Getting an analog input is a two-step process: sampling and conversion. Sampling requires a minimum time to allow the sampling capacitor to stabilize its voltage. Once the sampling terminates, the capacitor is isolated from the input so its voltage does not change during conversion. The conversion process is performed by a Successive Approximation Register (SAR) ADC which carries out a 10-step binary search, comparing the capacitor voltage to a new reference voltage at each step.

- The ADC provides a huge array of options which are only touched on in this chapter. The sample code in this chapter provides a manual method for taking a single ADC reading in the range 0-3.3 V in just over 2 μs. For details on how to use other reference value ranges, sample and convert in the background and use interrupts to announce the end of a sequence of conversions, etc., consult the Reference Manual.

10.5 Exercises

1. Configure the ADC for manual sampling and automatic conversion. Set Tad and the sampling time as short as possible while still meeting the minimum constraints.
2. Assume that the ADC is configured for manual sampling and automatic conversion. Write a function that begins sampling from a specified ANx pin, waits for the conversion to complete, and returns the result. This function will be useful whenever you need to take an ADC reading.
3. Using the configuration code and the ADC reading function you wrote for the previous questions, write a program that prompts the user to press ENTER and then reports the voltage on AN5 (in both ADC ticks and in volts) over the UART. Test the program with a variety of voltage dividers or a potentiometer.

Further Reading

PIC32 family reference manual. Section 17: 10-Bit analog-to-digital converter (ADC). (2011). Microchip Technology Inc.