Part III

Peripheral Reference

Chapter 7

Digital Input and Output

Digital inputs and outputs (DIO) are the simplest interfaces between the PIC32 and other electronics, sensors, and actuators. The PIC32 has many DIO pins, each of which normally has one of two states: high (1) or low (0). These states correspond to voltages in a circuit: usually 3.3 V or 0 V.[1] The DIO peripheral also handles change notification (CN) interrupts, which happen when the input changes on at least one of up to 22 digital inputs.

7.1 Overview

The PIC32 offers many DIO pins, arranged into "ports" A through G. The pins are labeled Rxy, where x is the port letter and y is the pin number. For example, pin 5 of port B is named RB5. Ports B and D are full 16-bit ports, with pins 0-15. (Port B can also be used for analog input.) Other ports have a smaller number of pins, not necessarily sequentially numbered; for example, port C has pins 1-4 and 12-15. All pins labeled Rxy can be used for input or output, except for RG2 and RG3, which are input only. For more details on the available pin numbers, see the Data Sheet.

The PIC32 has two types of digital outputs: buffered and open-drain. Usually, outputs are buffered and can drive the associated pin to either 0 V or 3.3 V. Some pins, however, can be configured with open-drains. Their pins can be driven to 0 V or to a high impedance state often called "floating" or "high-Z". When floating, the output is effectively disconnected, allowing you to attach an external "pull-up" resistor from the output to a positive voltage. Then, when the output floats, the external pull-up resistor connects the pin to the positive voltage (Figure 7.1).The positive voltage must be less than either 3.3 V or 5 V, depending on the pin (see, e.g., Figure 2.1 and Table 2.2 to determine which pins can tolerate 5 V).

A pin configured as an output should not source or sink more than about 10 mA. For example, it would be a mistake to connect a digital output to a 10 Ω resistor to ground. In this case, creating a digital high output of 3.3 V would require 3.3 V/10 Ω = 330 mA, which a digital output cannot source. Be careful, drawing too much current from a pin may damage your PIC32!

An input pin will read low, or 0, if the input voltage is close to zero, and it will read high, or 1, if it is close to 3.3 V. Some pins tolerate inputs up to 5 V. Some input pins, those that can also be used for "change notification" (labeled CNy, y = 0 ... 21, spread out over several of the ports), can be configured with an internal pull-up resistor to 3.3 V. If configured this way, the input will read "high" if it is disconnected (Figure 7.1). Otherwise, if an input pin is not connected to anything, we cannot be certain what the input will read.

Input pins have fairly high input impedance—very little current will flow in or out of an input pin. Therefore, connecting an external circuit to an input pin should have little effect on the behavior of the external circuit.

Up to 22 inputs can be configured as change notification inputs. When enabled, the change notification pins generate interrupts if their digital input state changes. The ISR must then read from the ports configured

[1] Technically, there are tolerances associated with these voltages, see the Data Sheet.

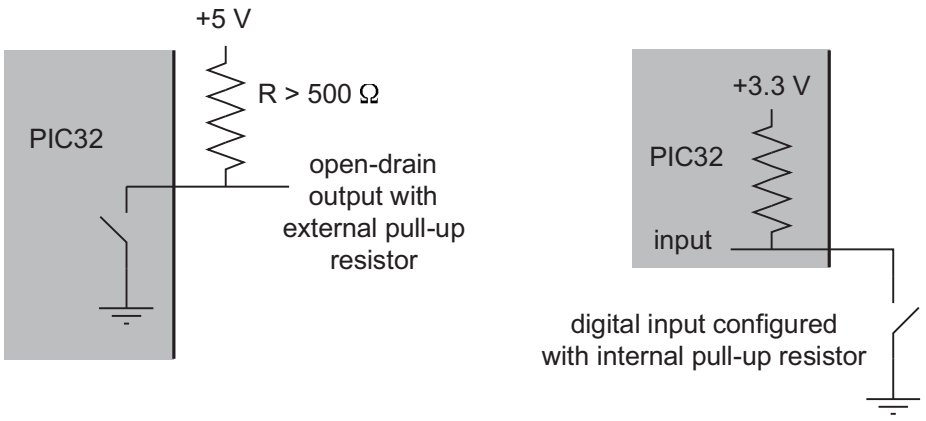


Figure 7.1: Left: A digital output configured as an open-drain output with external pull-up resistor to 5 V. (This should only be done for 5 V tolerant pins.) The resistor should allow no more than 10 mA to flow into the PIC32 when the PIC32 holds the output low. If the LAT bit controlling the pin has the value 1, the internal switch is open, and the output reads 5 V. If the bit is a 0, the internal switch is closed and the output reads 0 V. Right: A digital input configured with an internal pull-up resistor allows a simple open-close switch to yield digital high when the switch is open and digital low when the switch is closed.

with change notification, or else future input changes will not result in an interrupt. The ISR can compare the new port values to the previous values to determine which specific input has changed.

Microchip recommends that unused digital I/O pins be configured as outputs and driven to a logic low state, though this is not required. All pins are configured as inputs by default. The pins on port B, however, are shared with the analog-to-digital converter (ADC) and must be explicitly set to be digital (as opposed to analog) inputs.

7.2 Details

TRISx, x = A to G These tri-state SFRs determine which port x DIO pins are inputs and which are outputs. Bit y corresponds to the port's pin y (i.e., Rxy). Bits can be accessed individually by using TRISxbits.TRISxy. For example, TRISAbits.TRISA5 = 0 makes RA5 an output (0 = Output), and TRISAbits.TRISA5 = 1 makes RA5 an input (1 = Input). Bits of TRISx default to 1 on reset.

LATx, x = A to G A write to the latch chooses the output for pins configured as digital outputs. (Pins configured as inputs ignore the write.) Bit y correspond to that port's pin (i.e., Rxy). For example, if TRISD = 0x0000, making all 16 pins on port D outputs, then LATD = 0x00FF sets pins RD0-7 high and pins RD8-15 low. Individual pins can be referenced using LATxbits.LATxy, where y is the pin number. For example, LATDbits.LATD13 = 1 sets pin RD13 high. A write of 1 to an open-drain output sets the output to floating, while a write of 0 sets the output to low.

PORTx, x = A to G PORTx returns the current digital value for DIO pins on port x configured as inputs. Bit y corresponds to pin Rxy. Individual pins can be accessed as PORTxbits.RDy; for example, PORTDbits.RD6 returns the digital input for RD6.

ODCx, x = A to G The open-drain configuration SFR determines whether outputs are open drain or not. Individual bits can be accessed using ODCxbits.ODCxy. For example, if TRISbits.TRISD8 = 0, making RD8 an output, then ODCDbits.ODCD8 = 1 configures RD8 as an open-drain output, and ODCDbits.ODCD8 = 0 configures RD8 as a typical buffered output. The reset default for all bits is 0.

AD1PCFG The bits in this register determine whether port B's pins are analog or digital inputs. See Ch. 10 for information about analog inputs).

AD1PCFG⟨x⟩, or AD1PCFGbits.PCFGx, x = 0 to 15, controls whether pin ANx (equivalently RBx) is an analog input: 0 = analog input, 1 = digital pin.

Each of the lower 16 bits of this SFR correspond to a port B pin. On reset, they are zero, meaning that all port B pins are analog inputs by default. Therefore, to use a port B pin as a digital input, you must explicitly set the appropriate pin in AD1PCFG.

For example, to use port B, pin 2 (RB2) as a digital input, set AD1PCFGbits.PCFG2 (AD1PCFG⟨2⟩) to one. This extra configuration step applies only to port B because the other ports do not overlap with analog inputs.

CNPUE Change notification pull-up enable allows you to enable an internal pull-up resistor on the change notification pins (CN0-CN21). Each bit in CNPUE⟨21:0⟩, when set, enables the pull-up and when clear disables it. Bit y corresponds to pin CNy. Individual bits can be accessed using CNPUEbits.CNPUEx. For example, if CNPUEbits.CNPUE2 = 1, then CN2/RB0 has the internal pull-up resistor enabled, and if CNPUEbits.CNPUE2 = 0, then it is disabled. The reset default for all bits is 0 so the pull-up resistors are disabled. An internal pull-up resistor can be convenient because it ensures that the input pin is in a known state when disconnected.

Latches and Ports: What is the difference between the latch (LATx) and port (PORTx) SFRs? The PORTx SFRs correspond to voltages on the pins while LATx SFRS correspond to what the pin should output if configured as an output. When you read from LATx you are actually reading the last value you wanted output on the port, not its actual state. Therefore, to actually read pins, use PORTx.
Setting a bitfield in C requires a read-modify-write operation. For example, `LATFbits.LATF3 = 1` cause the CPU to read LATF, set bit 13, and write the result to LATF. Likewise, `PORTFbits.RF3 = 1` does the same, except through PORTF. If the voltage on another output pin (say RF4) is not at the desired level—perhaps a capacitor must charge—the initial read is incorrect which makes the output incorrect. Writing to LATx avoids this scenario because the latch contains the desired, not actual, state.
You can entirely avoid read-modify-write operations by using LATxINV, LATxCLR, or LATxSET, which directly modify LATx in hardware. We recommend using these only when needed: the LATxbits.Rxy syntax is clearer and less error prone. **Bottom line: read from PORTs, write to LATs.**

7.2.1 Change Notification

Change notification interrupts can be generated on pins CN0 - CN21 and are triggered when the input state on the pin changes. The relevant SFRs are:

CNPUE Change notification pull-up enable. See above (it is listed there because pull-ups can be useful even if no change notification is used).

CNCON The change notification control SFR enables CN interrupts if CNCON⟨15⟩ (or CNCONbits.ON) equals 1. The default is 0.

CNEN A particular pin CNx can generate a change notification interrupt if CNEN⟨x⟩ (or CNENbits.CNENx) is 1. Otherwise it is not included in the change notification.

The relevant interrupt bit fields and constants are:

IFS1⟨0⟩ or IFS1bits.CNIF: Interrupt status flag for change notification, set when the interrupt is pending.

IEC1⟨0⟩ or IEC1bits.CNIE: Interrupt enable flag for change notification, set to enable the interrupt.

IPC6⟨20:18⟩ or IPC6bits.CNIP: Change notification interrupt priority.

IPC6⟨17:16⟩ or IPC6bits.CNIS: Change notification sub-priority.

Vector Number: The change notification uses interrupt vector 26 or `_CHANGE_NOTICE_VECTOR`

A recommended procedure for enabling the CN interrupt:

1. Write an ISR using the vector `_CHANGE_NOTICE_VECTOR` (26). It should clear IFS1bits.CNIF and reads the pins involved in the CN scan to re-enable the interrupt.
2. Disable all interrupts using `__builtin_disable_interrupts()`
3. and CNENbits.CNENx to one, for each pin x that you want included in the change notification.
4. Set CNCONbits.ON to one.
5. Set the interrupt priority IPC6bits.CNIP and subpriority IPC6bits.CNIS.
6. Clear the interrupt status flag IFS1bits.CNIF.
7. Enable the CN interrupt by setting IEC1bits.CNIE to one.
8. Enable interrupts using `__builtin_enable_interrupts()`

7.3 Sample Code

Our first program, `simplePIC.c`, demonstrated the use of two digital outputs (to control two LEDs) and one digital input (the USER button).

The example below configures the following inputs and outputs:

- Pins RB0 and RB1 are digital inputs with internal pull-up resistors enabled.
- Pins RB2 and RB3 are digital inputs without pull-ups.
- Pins RB4 and RB5 are buffered digital outputs.
- Pins RB6 and RB7 are open-drain digital outputs.
- Pins AN8–AN15 are analog inputs.
- All port F pins are digital inputs and RF4 uses an internal pull-up.
- Change notification is enabled on pins RB0 (CN2), RF4 (CN17), and RF5 (CN18). Since both ports B and F are involved in the change notification, both ports must be read inside the ISR to allow the interrupt to be re-enabled. The ISR toggles one of the NU32 LEDs to indicate that a change has been noticed on pin RB0, RF4, or RF5.

Code Sample 7.1. `DIO_sample.c`. Digital input, output, and change notification.

```
//#define NU32_STANDALONE  // uncomment if program is standalone, not bootloaded
#include "NU32.h"          // peripheral.h, config bits, constants, funcs for startup and UART

unsigned int oldB = 0, oldF = 0, newB = 0, newF = 0;   // to hold the port B and F reads

void __ISR(_CHANGE_NOTICE_VECTOR, IPL3SOFT) CNISR(void) { // INT step 1
  newB = PORTB;           // since pins on port B and F are being monitored
  newF = PORTF;           // by CN, must read both to allow continued functioning
                          // ... do something here with oldB, oldF, newB, newF ...
  oldB = newB;            // save the current values for future use
  oldF = newF;
  LATBINV = 0xF0;         // toggle buffered RB4, RB5 and open-drain RB6, RB7
  NU32_LED1 = !NU32_LED1; // toggle LED1
  IFS1CLR = 1;            // clear the interrupt flag
}
```

```
int main(void) {
  NU32_Startup(); // cache on, min flash wait, interrupts on, LED/button init, UART init

  AD1PCFG = 0x00FF;       // set port B pins 8-15 as analog in, 0-7 as digital pins
  TRISB = 0xFF0F;         // set port B pins 4-7 as digital outputs, 0-3 as digital inputs
  ODCBSET = 0x00C0;       // set ODCB bits 6 and 7, so RB6, RB7 are open drain outputs
  CNPUEbits.CNPUE2 = 1;   // CN2/RB0 input has internal pull-up
  CNPUEbits.CNPUE3 = 1;   // CN3/RB1 input has internal pull-up
  CNPUEbits.CNPUE17 = 1;  // CN17/RF4 input has internal pull-up

  oldB = PORTB;           // bits 0-3 are relevant input
  oldF = PORTF;           // all pins of port F are inputs, by default
  LATB = 0x0050;          // RB4 is buffered high, RB5 is buffered low,
                          // RB6 is floating open drain (could be pulled to 3.3 V by
                          // external pull-up resistor), RB7 is low

  __builtin_disable_interrupts(); // step 1: disable interrupts
  CNCONbits.ON = 1;               // step 2: configure peripheral: turn on CN
  CNENbits.CNEN2 = 1;             // Use CN2/RB0 as a change notification
  CNENbits.CNEN17 = 1;            // Use CN17/RF4 as a change notification
  CNENbits.CNEN18 = 1;            // Use CN18/RF5 as a change notification

  IPC6bits.CNIP = 3;              // step 3: set interrupt priority
  IPC6bits.CNIS = 2;              // step 4: set interrupt subpriority
  IFS1bits.CNIF = 0;              // step 5: clear the interrupt flag
  IEC1bits.CNIE = 1;              // step 6: enable the CN interrupt
  __builtin_enable_interrupts();  // step 7: CPU enabled for mvec interrupts

  while(1) {
      ;                           // infinite loop
  }
  return 0;
}
```

7.4 Chapter Summary

- The PIC32 has 7 DIO ports, labeled with the letters A through G. Only ports B and D have all 16 pins. Almost every pin can be configured as a digital input or a digital output. Some outputs can be configured to be open-drain.
- By default, port B inputs are configured as analog inputs. To use these pins as digital inputs you must set the corresponding bits in AD1PCFG to one.
- Twenty-two pins can be configured as change notification pins (CN0 - CN21). For those pins that are enabled as change notification pins, any change of input state generates an interrupt. To re-enable the interrupt, the ISR should both clear the IRQ flag as well as read the pins involved in the change notification.
- The change notification pins offer an optional internal pull-up resistor so that the input registers as high when it is left floating. These pull-up resistors can be used regardless of whether change notification is enabled for the pin. The internal pull-up resistor allows simple interfacing of push-buttons, for example.

7.5 Exercises

1. True or false? If an input pin is not connected to anything, it always reads digital low.

2. You are configuring port B to receive analog and digital inputs and to write digital output. Here is how you would like to configure the port. (Pin x corresponds to RBx.)

   - Pin 0 is an analog input.
   - Pin 1 is a "typical" buffered digital output.
   - Pin 2 is an open-drain digital output.
   - Pin 3 is a "typical" digital input.
   - Pin 4 is a digital input with an internal pull-up resistor.
   - Pins 5-15 are analog inputs.
   - Pin 3 is monitored for change notification, and the change notification interrupt is enabled.

   Questions:

   (a) Which digital pin would most likely have an external pull-up resistor? What would be a reasonable range of resistances to use? Explain what factors set the lower bound on the resistance and what factors set the upper bound on the resistance.

   (b) To achieve the configuration described above, give the eight-digit hex values you should write to AD1PCFG, TRISB, ODCB, CNPUE, CNCON, and CNEN. (Some of these SFRs have unimplemented bits 16-31; write 0 for those bits.)

Chapter 8

Counter/Timers

Counters count rising edges of a signal. The pulses may come from the internal peripheral bus clock or external sources. If a fixed frequency clock produces the pulses, counters become timers (the count represents a time). Therefore, the words counter and timer are often used interchangeably. Because Microchip's documentation refers to these devices as timers, we adopt that terminology. Timers can generate interrupts after a preset number of pulses has been counted or on the falling edge of an external pulse whose duration is being timed. These timers differ from the core timer introduced in 5 in that they are peripherals rather than part of the MIPS32 CPU.

8.1 Overview

The PIC32 is equipped with five 16-bit peripheral timers named Timerx, where x is 1-5. A timer increments on the rising edge of a clock signal, which may come from the PBCLK or from an external source of pulses. The input for an external source for Timerx is pin TxCK. A prescaler $N \geq 1$ determines how many clock pulses must be received before the timer increments. If the prescaler is set to $N = 1$, the timer increments on every clock rising edge; if it is set to $N = 8$, it increments every eighth rising edge. The clock source type (internal or external) and the prescaler value is chosen by setting the value of the SFR TxCON.

Each 16-bit timer can count from 0 up to a period $P \leq 2^{16} - 1 = 65535 = 0\text{xFFFF}$. The current count is stored in the SFR TMRx and the value of P can be chosen by writing to the period register SFR PRx. When the timer reaches the value P, a *period match* occurs, and after N more pulses are received, the counter "rolls over" to 0. If the input to the timer is the 80 MHz PBCLK, with 12.5 ns between rising edges, then the time between rollovers is $T = (P + 1) \times N \times 12.5$ ns. Choosing $N = 1$ and $P = 79,999$, we get $T = 1$ ms, and changing N to 8 gives $T = 8$ ms. By configuring the timer to use the PBCLK and to generate an interrupt when a period match occurs, the timer can implement a function that runs at a fixed frequency (a controller, for example). See Figure 8.1.

If the period P is zero, then once the count reaches zero it will never increment again (it keeps rolling over). No interrupt can be generated by a period match if $P = 0$.

The PIC32 has two types of timers: Type A and Type B, each with slightly different features (explained shortly). Timer1 is type A and Timers 2-5 are type B. The timers can be used in the following modes, chosen by the SFR TxCON:

Counting PBCLK pulses. In this mode, the timer counts PBCLK pulses, so the count corresponds to an elapsed time. This mode is often used to generate interrupts at a desired frequency by appropriate setting of N and P. It can also be used to time the duration of code, like how we used the core timer in Chapter 5 A peripheral timer, however, can increment once every N PBCLK cycles, including $N = 1$, not just every 2 SYSCLK cycles.

Synchronous counting of external pulses. For the timer Timerx, an external pulse source is connected to the pin TxCK. The timer count increments after each rising edge of the external source. This mode is called "synchronous" because timer increments are synchronized to the PBCLK; the timer does not

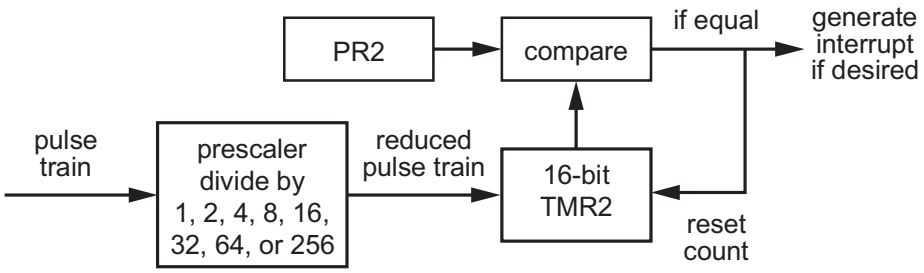


Figure 8.1: Simplified block diagram for a typical use of the 16-bit Timer2. The input pulses, which can be from the PBCLK or an external source, feeds a prescaler, which produces one output pulse for every N input pulses ($N = 1$, 2, 3, 8, 16, 32, 64, or 256). The TMR2 SFR stores the count of these pulses. TMR2 resets to zero on the first pulse after TMR2 matches the period register PR2. By default, PR2 is 0xFFFF, so TMR2 counts up to $2^{16} - 1$ before rolling over to zero. Timers 3,4, and 5 are similar to Timer 2, but Timer 1 can only have prescaler value $N = 1$, 8, 64, or 256.

actually increment until the first rising edge of PBCLK after the rising edge of the external source. If the external pulses are too fast, the timer will not accurately count them. According to the Electrical Characteristics section of the Data Sheet, the duration of the high and low portions of a pulse should be at least 37.5 ns each.

Asynchronous counting of external pulses (Type A Timer1 only). The Type A pulse counting circuit can be configured to increment independently of the PBCLK, allowing it to count even when the PBCLK is not operating, such as in the power-saving Sleep mode. If a period match occurs, Timer1 can generate an interrupt and wake up the PIC32. When used in asynchronous mode, the timer can count pulses with high and low durations as low as 10 ns each.

Timing the duration of an external pulse. Also called "gated accumulation mode." For Timerx, when the input on external pin TxCK goes high, the counter starts incrementing according to the PBCLK and the prescaler N. When the input drops low, the count stops. The timer can also generate an interrupt when the input drops low.

Important differences between Timer1 (Type A) and Timer2-5 (Type B) are:

- Only Timer1 can count external pulses asynchronously, as described above.
- Timer1 can have prescalers $N = 1$, 8, 64, and 256, while Timer2-5 can have prescalers $N = 1$, 2, 4, 8, 16, 32, 64, and 256.
- Timer2 and Timer3 can be chained together to make a single 32-bit timer, called Timer23. Timer4 and Timer5 can similarly be used to make a single 32-bit timer, called Timer45. These combined timers allow counts of up to $2^{32} - 1$, or over 4 billion. When two timers are used to make Timerxy (x < y), Timerx is called the "master" and Timery is the "slave"—only the prescaler and mode information in TxCON are relevant, while those fields in TyCON are ignored. When Timerx rolls over from $2^{16} - 1$ to 0, it sends a clock pulse to increment Timery. In Timerxy mode, the 16 bits of TMRy are copied to the most significant 16 bits of the SFR TMRx, so the 32 bits of TMRx contain the full count of Timerxy. Similarly, the 32 bits of PRx contain the 32-bit period match value Pxy. The interrupt associated with a period match (or a falling input in gated accumulation mode) is actually generated by Timery, so interrupt settings should be chosen for Timery's IRQ and vector.

Timers are used in conjunction with digital waveform generation by the Output Compare peripheral (Chapter 9) and in timing digital input waveforms by the Input Capture peripheral (Chapter 15). A timer can also be used to repeatedly trigger analog to digital conversions (Chapter 10).

8.2 Details

The following SFRs are associated with the timers. All SFRs default to 0x0000, except the PRx SFRs, which default to 0xFFFF. First, we describe settings common to both types of timers.

TxCON, x = 1 to 5 The Timerx control SFR configures the behavior of Timerx. Important bits common to both Type A and Type B timers include

TxCON⟨15⟩, or TxCONbits.ON: Enables and disables the timer.

1 Timerx enabled.

0 Timerx disabled.

TxCON⟨7⟩, or TxCONbits.TGATE: Sets gated accumulation mode, which can be used to time the duration of an external pulse. In gated accumulation mode the timer starts counting when an external signal goes high and stops when it goes low.

1 Gated accumulation mode enabled.

0 Gated accumulation mode disabled.

Gated accumulation also requires TxCONbits.TCS = 0. The gate input pin is TxCK.

TxCON⟨1⟩, or T1CONbits.TCS: Determines whether the timer uses an external or clock source or PBCLK.

1 Timerx uses the signal on TxCK as an external pulse source.

0 PBCLK provides the pulse source.

The Type A timer, Timer1, also has the relevant fields:

T1CON The control register for Timer1. Has the same fields as above; here we describe the fields specific to Timer1.

T1CON⟨5:4⟩, or T1CONbits.TCKPS: Sets the prescaler ratio. The prescaler determines how many pulses are required to increment the timer count. Type A timers have fewer prescaler ratios than Type B timers.

0b11 : Prescaler of 1:256.

0b10 : Prescaler of 1:64.

0b01 : Prescaler of 1:8.

0b00 : Prescaler of 1:1.

T1CON⟨2⟩, or T1CONbits.TSYNC: Determines whether external clock inputs are synchronized to PBCLK. When synchronized, the timer only registers a pulse once per PBCLK tick. When asynchronous, every external pulse is registered immediately (subject to timing requirements specified in the Data Sheet). Only Type A timers can count pulses asynchronously.

1 External signals are synchronized.

0 External signals are asynchronous.

The Type B timers, Timer2 - Timer5 have their own type-specific TxCON fields.

TxCON, x = 2 to 5 Here we describe the fields in TxCON specific to Type B timers.

TxCON⟨6:4⟩, or TxCONbits.TCKPS: Sets the prescaler ratio; there are many more choices than for Type A timers.

0b111 Prescaler of 1:256.

0b110 Prescaler of 1:64.

0b101 Prescaler of 1:32.

0b100 Prescaler of 1:16.

0b011 Prescaler of 1:8.

IRQ source	Vector	Flag	Enable	Priority	Sub-priority
Timer1	4 _TIMER_1_VECTOR	IFS0⟨4⟩ IFS0bits.T1IF	IEC0⟨4⟩ IEC0bits.T1IE	IPC1⟨4:2⟩ IPC1bits.T1IP	IPC1⟨1:0⟩ IPC1bits.T1IS
Timer2	8 _TIMER_2_VECTOR	IFS0⟨8⟩ IFS0bits.T2IF	IEC0⟨8⟩ IEC0bits.T2IE	IPC2⟨4:2⟩ IPC2bits.T2IP	IPC2⟨1:0⟩ IPC2bits.T2IS
Timer3	12 _TIMER_3_VECTOR	IFS0⟨12⟩ IFS0bits.T3IF	IEC0⟨12⟩ IEC0bits.T3IE	IPC3⟨4:2⟩ IPC3bits.T3IP	IPC3⟨1:0⟩ IPC3bits.T3IS
Timer4	16 _TIMER_4_VECTOR	IFS0⟨16⟩ IFS0bits.T4IF	IEC0⟨16⟩ IEC0bits.T4IE	IPC4⟨4:2⟩ IPC4bits.T4IP	IPC4⟨1:0⟩ IPC4bits.T4IS
Timer5	20 _TIMER_5_VECTOR	IFS0⟨20⟩ IFS0bits.T5IF	IEC0⟨20⟩ IEC0bits.T5IE	IPC5⟨4:2⟩ IPC5bits.T5IP	IPC5⟨1:0⟩ IPC5bits.T5IS

Table 8.1: Interrupt settings relevant to timers.

0b010 Prescaler of 1:4.

0b001 Prescaler of 1:2.

0b000 Prescaler of 1:1.

TxCON⟨3⟩, or TxCONbits.T32: This bit is only relevant for x = 2 and 4 (Timer2 and Timer4). When set, Timerx and Timery are chained together to make the 32 bit timer called Timerxy (y = x +1). When in 32-bit mode, TyCON settings are ignored, TMRy is enabled, and its clock value comes from the rollover of TMRx after it hits 0xFFFF. Interrupts are generated by Timery, but the timer's full 32-bit value and 32-bit rollover count are accessed via TMRx and PRx. When clear, Timerx and Timery operate as independent 16-bit timers.

1 Use Timer23 (if x = 2) or Timer45 (if x = 4) as 32-bit timers.

0 User Timerx as a 16-bit timer.

The following SFRs apply to all Timerx, x = 1 to 5.

TMRx, x = 1 to 5 TMRx⟨15:0⟩ stores the 16-bit count of Timerx. TMRx resets to 0 on the next count after TMRx reaches the number stored in PRx. This rollover process is called a period match. In Timerxy 32-bit mode, TMRx contains the 32-bit value of the chained timer, and period match occurs when TMRx = PRx.

PRx, x = 1 to 5 PRx⟨15:0⟩ contains the maximum value of the count of TMRx before it resets to zero on the next count. An interrupt can be generated on this period match. In Timerxy 32-bit timer mode, PRx contains the 32-bit value of the period Pxy. Interrupts are generated as if Timery were triggered the interrupt.

The timer can generate an interrupt on the falling edge of the gate input when it is in gated mode (TxCONbits.TCS = 0 and TxCONbits.TGATE = 1). Otherwise, it can generate an interrupt whenever a period match occurs.

The relevant interrupt flags are shown in Table 8.1. To enable the interrupt for Timerx, the interrupt enable bit IEC0bits.TxIE must be set. The interrupt flag bit IFS0bits.TxIF should be cleared and the priority and subpriority bits IPCxbits.TxIP and IPCxbits.TxIS must be written. In 32-bit Timerxy mode, interrupts are generated by Timery; interrupt settings for Timerx are ignored.

8.3 Sample Code

8.3.1 A Fixed Frequency ISR

To create a 5 Hz ISR with an 80 MHz PBCLK, the interrupt must be triggered every 16 million PBCLK cycles. The highest a 16-bit timer can count is $2^{16} - 1$. Instead of wasting two timers to make a 32-bit timer with a prescaler $N = 1$, let's use a single 16-bit timer with a prescaler $N = 256$. We'll use Timer1. We should choose PR1 to satisfy

```
16000000 = (PR1 + 1) * 256
```

that is, PR1 = 62499. The ISR in the following code toggles a digital output at 5 Hz, creating a 2.5 Hz square wave (a flashing LED on the NU32).

Code Sample 8.1. TMR_5Hz.c. Timer1 toggles RA5 five times a second (LED2 on the NU32 flashes).

```
//#define NU32_STANDALONE  // uncomment if program is standalone, not bootloaded
#include "NU32.h"          // config bits, constants, funcs for startup and UART

void __ISR(_TIMER_1_VECTOR, IPL5SOFT) Timer1ISR(void) {  // INT step 1: the ISR
  LATAINV = 0x20;                 // toggle RA5
  IFS0bits.T1IF = 0;              // clear interrupt flag
}

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

  __builtin_disable_interrupts(); // INT step 2: disable interrupts at CPU
                                  // INT step 3: setup peripheral
  PR1 = 62499;                    //             set period register
  TMR1 = 0;                       //             initialize count to 0
  T1CONbits.TCKPS = 3;            //             set prescaler to 256
  T1CONbits.TGATE = 0;            //             not gated input (the default)
  T1CONbits.TCS = 0;              //             PCBLK input (the default)
  T1CONbits.ON = 1;               //             turn on Timer1
  IPC1bits.T1IP = 5;              // INT step 4: priority
  IPC1bits.T1IS = 0;              //             subpriority
  IFS0bits.T1IF = 0;              // INT step 5: clear interrupt flag
  IEC0bits.T1IE = 1;              // INT step 6: enable interrupt
  __builtin_enable_interrupts();  // INT step 7: enable interrupts at CPU
  while (1) {
      ;                           // infinite loop
  }
}
```

8.3.2 Counting External Pulses

The following code uses the 16-bit timer Timer2 to count the rising edges on the input T2CK. The 32-bit Timer45 creates an interrupt at 2 kHz to toggle a digital output, generating a 1 kHz pulse train on RD1 that acts as input to T2CK. Although a 16-bit timer can certainly generate a 1 kHz pulse train, we use a 32-bit timer just to show the configuration. In Chapter 9 we will learn about the Output Compare peripheral, a better way to use a timer to create more flexible waveforms.

To create an IRQ every 0.5 ms (2 kHz), we use a prescaler $N = 1$ and a period match PR4 = 39,999, so

```
(PR4 + 1) * N * 12.5 ns = 0.5 ms
```

The code below displays to your computer's screen the amount of time that has elapsed since the PIC32 was reset, in milliseconds. If you wait 65 seconds, you'll see Timer2 roll over.

Code Sample 8.2. TMR_external_count.c. Timer45 creates a 1 kHz pulse train on RD1, and these external pulses are counted by Timer2. The elapsed time is periodically reported back to the host computer screen.

```
//#define NU32_STANDALONE  // uncomment if program is standalone, not bootloaded
#include "NU32.h"          // config bits, constants, funcs for startup and UART
```

```
void __ISR(_TIMER_5_VECTOR, IPL4SOFT) Timer5ISR(void) {  // INT step 1: the ISR
  LATDINV = 0b10;                                        // toggle RD1
  IFS0bits.T5IF = 0;                                     // clear interrupt flag
}

int main(void) {
  char message[200] = {};
  int i = 0;

  NU32_Startup();                 // cache on, interrupts on, LED/button init, UART init
  __builtin_disable_interrupts(); // INT step 2: disable interrupts

  TRISDbits.TRISD1 = 0;           // make D1 an output. connect D1 to C1/T2CK!

                                  // configure Timer2 to count external pulses.
                                  // The remaining settings are left at their defaults
  T2CONbits.TCS = 1;              // count external pulses
  PR2 = 0xFFFF;                   // enable counting to max value of 2^16 - 1
  TMR2 = 0;                       // set the timer count to zero
  T2CONbits.ON = 1;               // turn Timer2 on and start counting

                                  // 1 kHz pulses with 2 kHz interrupt from Timer45
  T4CONbits.T32 = 1;              // INT step 3: set up Timers 4 and 5 as 32-bit Timer45
  PR4 = 39999;                    //             rollover at 40,000; 80MHz/40k = 2 kHz
  TMR4 = 0;                       //             set the timer count to zero
  T4CONbits.ON = 1;               //             turn the timer on
  IPC5bits.T5IP = 4;              // INT step 4: priority for Timer5 (int goes with T5)
  IFS0bits.T5IF = 0;              // INT step 5: clear interrupt flag
  IEC0bits.T5IE = 1;              // INT step 6: enable interrupt
  __builtin_enable_interrupts();  // INT step 7: enable interrupts at CPU

  while (1) {
                                  // display the elapsed time in ms
    sprintf(message,"Elapsed time:  %u ms\r\n", TMR2);
    NU32_WriteUART1(message);

    for(i = 0; i < 10000000; ++i){// loop to delay printing
      _nop();                     // include nop so loop is not optimized away
    }
  }
  return 0;
}
```

8.3.3 Timing the Duration of an External Pulse

In this last example we modify our previous code so that Timer45 creates a train of pulses on RD1 that are high for one second and low for one second (a 0.5 Hz square wave). These pulses are timed by the 32-bit Timer23 in gated accumulation mode. The accumulated count begins when the T2CK input from RD1 goes high and stops when the T2CK input drops low. The falling edge calls an ISR that resets the Timer23 count and displays the count, and the duration in seconds, to the screen. You should find that the count is very close to 80 million, as expected for the one second pulses generated by Timer45.

Code Sample 8.3. `TMR_pulse_duration.c`. Timer45 creates a series of 1 second pulses on RD1. These pulses are input to T2CK and Timer23 measures their duration.

```
//#define NU32_STANDALONE  // uncomment if program is standalone, not bootloaded
#include "NU32.h"          // config bits, constants, funcs for startup and UART
```

```
void __ISR(_TIMER_5_VECTOR, IPL4SOFT) Timer5ISR(void) {  // INT step 1: the ISR
  LATDINV = 0b10;                 // toggle RD1
  IFS0bits.T5IF = 0;              // clear interrupt flag
}

void __ISR(_TIMER_3_VECTOR, IPL3SOFT) Timer23ISR(void) { // INT step 1: the ISR
  char msg[100] = {};

  sprintf(msg,"The count was %u, or %10.8f seconds.\r\n", TMR2, TMR2/80000000.0);
  NU32_WriteUART1(msg);
  TMR2 = 0;                       // reset Timer23
  IFS0bits.T3IF = 0;              // clear interrupt flag
}

int main(void) {
  NU32_Startup();                 // cache on, interrupts on, LED/button init, UART init

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

  TRISDbits.TRISD1 = 0;           // make D1 an output. connect D1 to C1/T2CK!

                                  // INT step 3
  T2CONbits.T32 = 1;              // for T23:    combine Timers 2 and 3 to make Timer23
  T2CONbits.TGATE = 1;            //             Timer23 in gated accumulation mode
  PR2 = 0xFFFFFFFF;               //             use the full period of Timer23
  T2CONbits.TON = 1;              //             turn Timer23 on
  T4CONbits.T32 = 1;              // for T45:    enable 32 bit mode Timer45
  PR4 = 79999999;                 //             set PR so timer rolls over at 1 Hz
  TMR4 = 0;                       //             initialize count to 0
  T4CONbits.TON = 1;              //             turn Timer45 on
  IPC5bits.T5IP = 4;              // INT step 4: priority for Timer5 (int for Timer45)
  IPC3bits.T3IP = 3;              //             priority for Timer3 (int for Timer23)
  IFS0bits.T5IF = 0;              // INT step 5: clear interrupt flag for Timer45
  IFS0bits.T3IF = 0;              //             clear interrupt flag for Timer23
  IEC0bits.T5IE = 1;              // INT step 6: enable interrupt for Timer45
  IEC0bits.T3IE = 1;              //             enable interrupt for Timer23
  __builtin_enable_interrupts();  // INT step 7: enable interrupts at the CPU
  while (1) {
    ;
  }
}
```

8.4 Chapter Summary

- All five of the 16-bit PIC32 timers can be used to generate fixed-frequency interrupts, count external pulses, and time the duration of external pulses. Additionally, the Type A Timer1 can asynchronously count external pulses even when the PIC32 is in Sleep mode, while the Type B timers Timer2 and Timer3 can be chained to make the 32-bit timer Timer23. Similarly, Timer4 and Timer5 can be chained to make the 32-bit timer Timer45.

- For a 32-bit timer Timerxy, the timer configuration information in TxCON is used (TyCON is ignored), and the interrupt enable, flag status, and priority bits are configured for Timery (this information for Timerx is ignored). The 32-bit Timerxy count is held in TMRx and the 32-bit period match value is held in PRx.

- A timer can generate an interrupt when either the external pulse being timed falls low (gated accumulation mode) or the count reaches a value stored in a period register (period match).

8.5 Exercises

1. Assume PBCLK is running at 80 MHz. Give the four-digit hex values for T3CON and PR3 so that Timer3 is enabled, accepts PBCLK as input, has a 1:64 prescaler, and rolls over (generates an interrupt) every 16 ms.

2. Using a 32-bit timer (Timer23 or Timer45), what is the longest duration you can time, in seconds, before the timer rolls over? (Use the prescaler that maximizes this time.)

Chapter 9

Output Compare

The output compare (OC) peripheral sets the state of an output pin based on the value of a timer. Output compare can be used to generate a single pulse of specified duration or a continuous pulse train. Either mode of operation can generate an interrupt when the value of the output pin changes.

Like most microcontrollers, the PIC32 cannot output an arbitrary analog voltage because it lacks a true digital-to-analog converter (DAC) (See Ch. 16 for details about the PIC32's limited analog output capability). By generating a pulse train, the output compare can be used to generate analog signals. The analog value is proportional to the *duty cycle* of the pulse train: the percentage of the period that the signal is high. Generating a signal whose value is determined by the duty cycle is called "pulse width modulation" (PWM) (see Fig. 9.1). PWM signals are commonly used as input to H-bridge amplifiers that drive motors. High-frequency PWM signals can also be low-pass filtered to create a true analog output.

9.1 Overview

The PIC32's five OC peripherals can be configured to operate in seven different modes. In every mode, the module uses either the count of the 16-bit timer Timer2 or Timer3, or the count of the 32-bit timer Timer23, depending on the OC control SFR OCxCON (where x = 1...5 refers to the particular output compare module). We call the timer used by an output compare module Timery, where y = 2, 3, or 23. You must configure Timery with its own prescaler and period register, which influences the OC peripheral's behavior.

The OC peripheral's operating modes consist of three "single compare" modes, two "dual compare" modes, and two PWM modes. In the single compare modes, the Timery count TMRy is compared to the value in OC count SFR OCxR. When the counts match, the OC output is either set high, cleared low, or toggled,

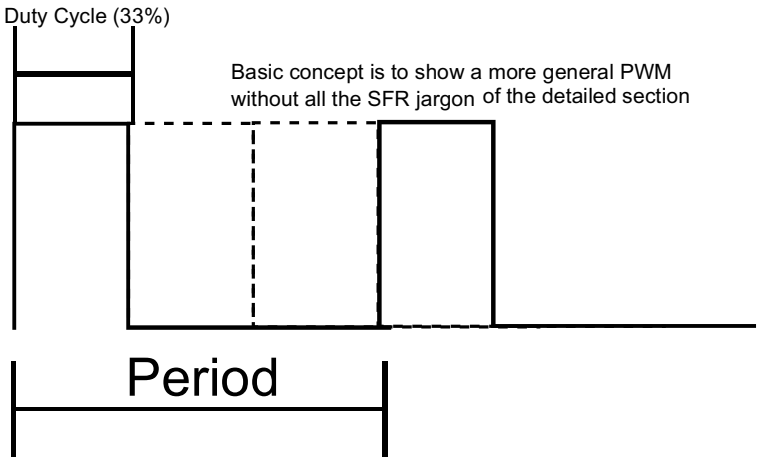


Figure 9.1: A PWM waveform. The duty cycle is the percentage of a period that the signal is high.

depending on the mode. Toggle mode generates a continuous pulse train, with the period determined by the Timery period register PRy and the pulse duration determined by OCxR.

In the dual compare modes, TMRy is compared to two values, OCxR and OCxRS. When TMRy matches OCxR the output is driven high, and when it matches OCxRS it is driven low. Depending on a bit in OCxCON, either a single pulse or continuous pulse train is produced.

The two PWM modes create continuous pulse trains. Each pulse begins (is set high) at rollover of Timery, as set by the period register PRy. The output is then set low when the timer count reaches OCxR. To change the value of OCxR, the user's program may alter the value in OCxRS at any time. This value will then be transferred to OCxR at the beginning of the new time period. The duty cycle of the pulse train, as a percentage, is

```
duty cycle = OCxR/(PRy + 1) x 100%.
```

One of the two PWM modes offers the use of a fault protection input. If chosen, the OCFA input pin (corresponding to OC1 through OC4) or the OCFB input pin (corresponding to OC5) must be high for PWM to operate. If the pin drops to logic low, corresponding to some external fault condition, the PWM output will be high impedance (like an open-drain output, effectively disconnected) until the fault condition is removed and the PWM mode is reset by a write to OCxCON.

9.2 Details

The output compare modules are controlled by the following SFRs. The OCxCON SFRs default to 0x0000 on reset; the OCxR and OCxRS SFR values are unknown after reset.

OCxCON, x = 1 to 5 This output compare control SFR determines the operating mode of OCx.

OCxCON⟨15⟩, or OCxCONbits.ON: Enables and disables the output compare module.

- 1 Output compare enabled.
- 0 Output compare disabled.

OCxCON⟨5⟩, or OCxCONbits.OC32: Determines which timer to use.

- 1 Use the 32-bit timer Timer23.
- 0 Use a 16-bit timer, either Timer2 or Timer3.

OCxCON⟨4⟩, or OCxCONbits.OCFLT: The read-only PWM fault condition status bit. If a fault has occurred you must reset the PWM module by writing to OCxCONbits.OCM (assuming the external fault condition has been removed).

- 1 PWM fault has occurred.
- 0 No fault has occurred.

OCxCON⟨3⟩, or OCxCONbits.OCTSEL: This timer select bit chooses the timer used for comparison. If using the 32-bit Timer23, then this bit is ignored.

- 1 Use Timer3.
- 0 Use Timer2.

OCxCON⟨2:0⟩, or OCxCONbits.OCM: These three bits determine the operating mode:

0b111 PWM mode with fault pin enabled. OCx is set high on the timer rollover, then set low when the timer value matches OCxR. The SFR OCxRS can be altered at any time, and is copied to OCxR at the beginning of the next timer period.[1] The duty cycle of the PWM signal is

$$\text{OCxR}/(\text{PRy} + 1) \times 100\%, \tag{9.1}$$

where PRy is the period register of the timer.

[1] Initialize OCxR before enabling the OC module to handle the first PWM cycle. After enabling the OC module, OCxR is read-only.

If the fault pin (OCFA for OC1-4 and OCFB for OC5) drops low, the read-only fault status bit OCxCONbits.OCFLT is set to 1, the OCx output is set to high impedance, and an interrupt is generated. The fault condition is cleared and PWM resumes once the fault pin goes high and the OCxCONbits.OCM bits are rewritten.
You can use the fault pin with an Emergency Stop button that is normally high but drops low when the user presses it. If the OCx output is driving an H-bridge that powers a motor, setting the OCx output to high impedance will signal the H-bridge to stop sending current to the motor. An emergency stop button will likely have other requirements (such as also physically cutting power to the motor), depending on your application.

0b110 PWM mode with fault pin disabled. Identical to above, except without the fault pin.

0b101 Dual compare mode, continuous output pulses. When the module is enabled, OCx is driven low. OCx is driven high on a match with OCxR and driven low on a match with OCxRS. The process repeats, creating an output pulse train. An interrupt can be generated when OCx is driven low.

0b100 Dual compare mode, single output pulse. Same as above, except the OCx pin will remain low after the match with OCxRS until the OC mode is changed or the module is disabled.

0b011 Single compare mode, continuous pulse train. When the module is enabled, OCx is driven low. The output is toggled on all future matches with OCxR, until the mode is changed or the module disabled. Each toggle can generate an interrupt.

0b010 Single compare mode, single high pulse. When the module is enabled, OCx is driven high. OCx will be driven low and an interrupt optionally generated on a match with OCxR. OCx remains low until the mode is changed or the module disabled.

0b001 Single compare mode, single low pulse. When the module is enabled, OCx is driven low. OCx will be driven high and an interrupt optionally generated on a match with OCxR. OCx will remain high until the mode is changed or the module disabled.

0b000 The output compare module is disabled but still drawing current, unless OCxCONbits.ON = 0.

OCxR, x = 1 to 5 If OCxCONbits.OC32 = 1, then all 32-bits of OCxR are compared against Timer23's 32-bit count. Otherwise, only OCxR⟨15:0⟩ is compared to the 16-bit count of Timer2 or Timer3, depending on OCxCONbits.OCTSEL.

OCxRS, x = 1 to 5 In dual compare mode, if OCxCONbits.OC32 = 1, then all 32-bits of OCxRS are compared against Timer23's 32-bit count. Otherwisee, only OCxRS⟨15:0⟩ is compared to the 16-bit counter Timer2 or Timer3, depending on OCxCONbits.OCTSEL. In PWM mode, the value of this register is transferred into OCxR at the beginning of each period; therefore, modifying this register sets the next duty cycle. This SFR is unused in the single compare modes.

Timer2, Timer3, or Timer23 (depending on OCxCONbits.OC32 and OCxCONbits.OCTSEL) must be separately configured. Output compare modules do not affect the behavior of the timers; they simply compare values in OCxR and OCxRS and alter the digital output OCx on match events.

The interrupt flag status and enable bits for OCx are IFS0bits.OCxIF and IEC0bits.OCxIE, and the priority and subpriority bits are IPCxbits.OCxIP and IPCxbits.OCxIS.

PWM Modes The most common modes for Output Compare are the PWM modes. They can be used to drive H-bridges powering motors or to continuously transmit analog values represented by the duty cycle. Microchip often refers to the duty cycle as the duration OCxR of the high portion of the PWM waveform, but it is more standard to refer to the duty cycle in the range 0 to 100%. A plot of a PWM waveform is shown in Figure 9.2.

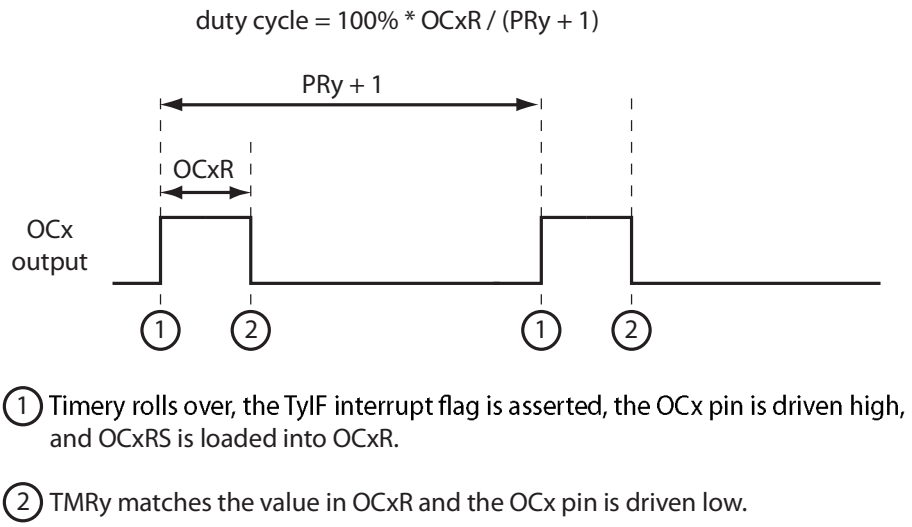


Figure 9.2: A PWM waveform from OCx using Timery as the time base.

9.3 Sample Code

9.3.1 Generating a Pulse Train with PWM

Below is sample code using OC1 with Timer2 to generate a 10 kHz PWM signal, initially at 25% duty cycle and then changed to 50% duty cycle. The fault pin is not used.

Code Sample 9.1. `OC_PWM.c` Generating 10 kHz PWM with 50% duty cycle.

```
//#define NU32_STANDALONE  // uncomment if program is standalone, not bootloaded
#include "NU32.h"          // config bits, constants, funcs for startup and UART

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

  T2CONbits.TCKPS = 2;     // Timer2 prescaler N=4 (1:4)
  PR2 = 1999;              // period = (PR2+1) * N * 12.5 ns = 100 us, 10 kHz
  TMR2 = 0;                // initial TMR2 count is 0
  OC1CONbits.OCM = 0b110;  // PWM mode without fault pin; other OC1CON bits are defaults
  OC1RS = 500;             // duty cycle = OC1RS/(PR2+1) = 25%
  OC1R = 500;              // initialize before turning OC1 on; afterward it is read-only
  T2CONbits.ON = 1;        // turn on Timer2
  OC1CONbits.ON = 1;       // turn on OC1

  _CP0_SET_COUNT(0);       // delay 4 seconds so you can see the 25% duty cycle on a 'scope
  while(_CP0_GET_COUNT() < 4 * 40000000) {
    ;
  }
  OC1RS = 1000;            // set duty cycle to 50%
  while(1) {
    ;                      // infinite loop
  }
  return 0;
}
```

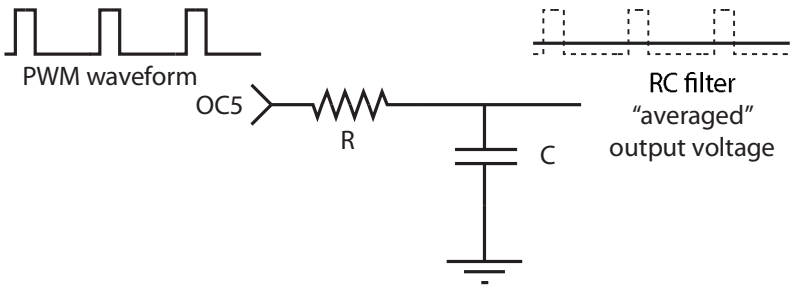


Figure 9.3: An RC low-pass filter "averaging" the PWM output from OC3.

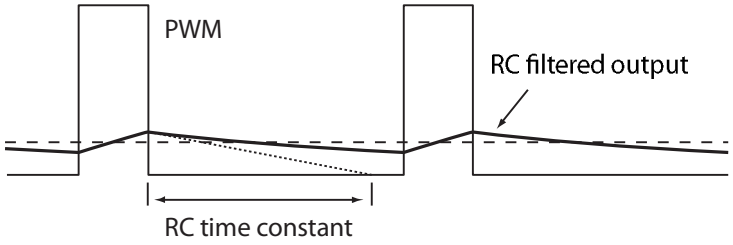


Figure 9.4: A close-up of the PWM, the RC filter output (with RC charging/discharging time constant illustrated), and the true time-averaged output (dashed). If the variation in the RC filtered output is unacceptably large, a larger value of RC should be chosen.

9.3.2 Analog Output

DC Analog Output

Low-pass filtering a constant duty cycle PWM signal results in a DC (constant) analog output. The low-pass filter essentially time-averages the high and low voltages of the waveform,

```
average voltage = duty cycle * 3.3 V
```

assuming that the output compare module swings between 0 and 3.3 V (the range may actually be a bit less).

There are many ways to build circuits to low-pass filter a signal, including *active* filter circuits using op amps. Here we focus on a simple *passive* RC filter, shown in Figure 9.3. The voltage V_C across the capacitor C is the output of the filter. When R is zero, the output compare module attempts to source or sink enough current to allow the capacitor voltage to exactly track the nominal PWM square wave, and there is no "averaging" effect. As the resistance R is increased, however, the resistor increasingly limits the current I available to charge or discharge the capacitor, meaning that the capacitor's voltage changes more and more slowly, according to the relationship $dV_C/dt = I/C$.

The charging and discharging of the capacitor, and its relationship to the product RC, is shown in Figure 9.4. RC low-pass filters are discussed in more detail in Appendix B.2.

In Figure 9.4, the RC filter voltage variation during one PWM cycle is rather large. To reduce this variation, we would choose a larger product RC by increasing the resistance and/or capacitance. The drawback of a large RC is that the filter's output voltage changes slowly in response to a change in the PWM duty cycle. We address this issue in more detail below.

The PWM OCxR value can range from 0 to PRy + 1, where PRy is the period register of the Timery base for the OCx module. This means that PRy + 2 different average voltage levels are achievable.

Time-Varying Analog Output

Suppose we want to create a sinusoidal analog output voltage by changing the duty cycle of the PWM. Let's call the frequency of this desired analog output f_a. Now we have three relevant frequencies: the PWM frequency f_{PWM}, the RC filter cutoff frequency $f_c = 1/(2\pi RC)$ (Appendix B.2.2), and the desired analog voltage frequency f_a. Examining the frequency response of the low-pass RC filter in Figure B.8(a), we can adopt the following rules of thumb for choosing these three frequencies:

- $f_{PWM} \geq 100 f_c$: The PWM waveform consists of a base frequency at f_{PWM} plus higher harmonics to create the square wave output. According to the gain response of the filter, only about 1% of the magnitude of the PWM frequency component at $100 f_c$ makes it through the RC filter. This is probably acceptable.
- $f_c \geq 10 f_a$: Again consulting the RC filter frequency response, we see that signals at 10 times less than f_c are relatively unaffected by the RC filter: the phase delay is only a few degrees and the gain is nearly 1.

For example, if the PWM is at 100 kHz, then we might choose an RC filter cutoff frequency of 1 kHz, and the highest frequency analog output we should expect to be able to create would be 100 Hz. In other words, we can vary the PWM duty cycle through a full sinusoid (e.g., from 50% duty cycle to 100% duty cycle to 0% duty cycle and back to 50% duty cycle) 100 times per second.[2] If the desired analog output is not sinusoidal, then it should be a sum of signals at frequencies less than 100 Hz.

The maximum possible PWM frequency is determined by the 80 MHz PBCLK and the number of bits of resolution we require for the analog output. For example, if we want 8 bits of resolution in the analog output levels, this means we need $2^8 = 256$ different PWM duty cycles. Therefore the maximum PWM frequency is 80 MHz / 256 = 312.5 kHz.[3] On the other hand, if we require $2^{10} = 1024$ voltage levels, the maximum PWM frequency is 78.125 kHz. Thus there is a fundamental trade-off between the voltage resolution and the maximum PWM frequency (and therefore the maximum analog output frequency f_a). While higher resolution analog output is generally desirable, there are limits to the value of increasing resolution beyond a certain point, because the device receiving the analog input may have a limit to its analog input sensing resolution and the transmission lines for the analog signal may be subject to electromagnetic noise that creates voltage variations larger than the analog output resolution.

The code below generates a PWM signal at 78.125 kHz with a duty cycle determined by OC3R in the range 0 to 1024. The timer base is Timer2. With a suitable resistor and capacitor attached to OC3, the voltage across the capacitor reflects the analog voltage requested by the user. Because the voltage does not change quickly in this example, you can choose f_c significantly lower than 781.25 Hz.

Code Sample 9.2. `OC_analog_out.c` Using Timer2, OC3, and an RC low-pass filter to create analog output.

```
//#define NU32_STANDALONE  // uncomment if program is standalone, not bootloaded
#include "NU32.h"          // config bits, constants, funcs for startup and UART

#define PERIOD 1024        // this is PR2 + 1
#define MAXVOLTAGE 3.3     // corresponds to max high voltage output of PIC32

int getUserPulseWidth(void) {
  char msg[100] = {};
  float f = 0.0;

  sprintf(msg, "Enter the desired voltage, from 0 to %3.1f (volts): ", MAXVOLTAGE);
  NU32_WriteUART1(msg);
```

[2] Note that this creates a signal that is the sum of a 100 Hz sinusoid with a duty cycle amplitude equal to 50% plus a DC (zero frequency) component of amplitude equal to 50% duty cycle.

[3] Technically this yields 257 possible duty cycle levels, since OCxR = 0 corresponds to 0% duty cycle and OCxR = 256 corresponds to 100% duty cycle.

```
  NU32_ReadUART1(msg,10);
  sscanf(msg, "%f", &f);

                                  // clamp the input voltage to the appropriate range
  if (f > MAXVOLTAGE) {
    f = MAXVOLTAGE;
  } else if (f < 0.0) {
    f = 0.0;
  }

  sprintf(msg, "\r\nCreating %5.3f volts.\r\n", f);
  NU32_WriteUART1(msg);
  return PERIOD * (f / MAXVOLTAGE);  // convert volts to counts
}

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

  PR2 = PERIOD - 1;       // Timer2 is the base for OC3, PR2 defines PWM frequency, 78.125 kHz
  TMR2 = 0;               // initialize value of Timer2
  T2CONbits.ON = 1;       // turn Timer2 on, all defaults are fine (1:1 divider, etc.)
  OC3CONbits.OCTSEL = 0;  // use Timer2 for OC3
  OC3CONbits.OCM = 0b110; // PWM mode with fault pin disabled
  OC3CONbits.ON = 1;      // Turn OC3 on
  while (1) {
    OC3RS = getUserPulseWidth();
  }
  return 0;
}
```

9.4 Chapter Summary

- Output Compare modules pair with Timer2, Timer3, or the 32-bit Timer23 to generate a single timed pulse or a continuous pulse train with controllable duty cycle. Microcontrollers commonly control motors using pulse-width modulation (PWM) to drive H-bridge amplifiers that power the motors.
- Low-pass filtering of PWM signals, perhaps using an RC filter with a cutoff frequency $f_c = 1/(2\pi RC)$, allows the generation of analog outputs. There is a fundamental tradeoff between the resolution of the analog output and the maximum possible frequency component f_a of the generated analog signal. If the PWM frequency is f_{PWM}, then generally the frequencies should satisfy $f_{PWM} \gg f_c \gg f_a$.

9.5 Exercises

1. Enforce the constraints $f_{PWM} \geq 100 f_c$ and $f_c \geq 10 f_a$. Given that PBCLK is 80 MHz, provide a formula for the maximum f_a given that you require n bits of resolution in your DC analog voltage outputs. Provide a formula for RC in terms of n.
2. You will use PWM and an RC low-pass filter to create a time-varying analog output waveform that is the sum of a constant offset and two sinusoids of frequency f and kf, where k is an integer greater than 1. The PWM frequency will be 10 kHz and f satisfies 50 Hz $\geq f \geq$ 10 Hz. Use OC1 and Timer2 to create the PWM waveform, and set PR2 to 999 (so the PWM waveform is 0% duty cycle when OC1R = 0 and 100% duty cycle when OC1R = 1000). You can break this program into the following pieces:
   (a) Write a function that forms a sampled approximation of a single period of the waveform

$$V_{out}(t) = C + A_1 \sin(2\pi f t) + A_2 \sin(2\pi k f t + \phi),$$

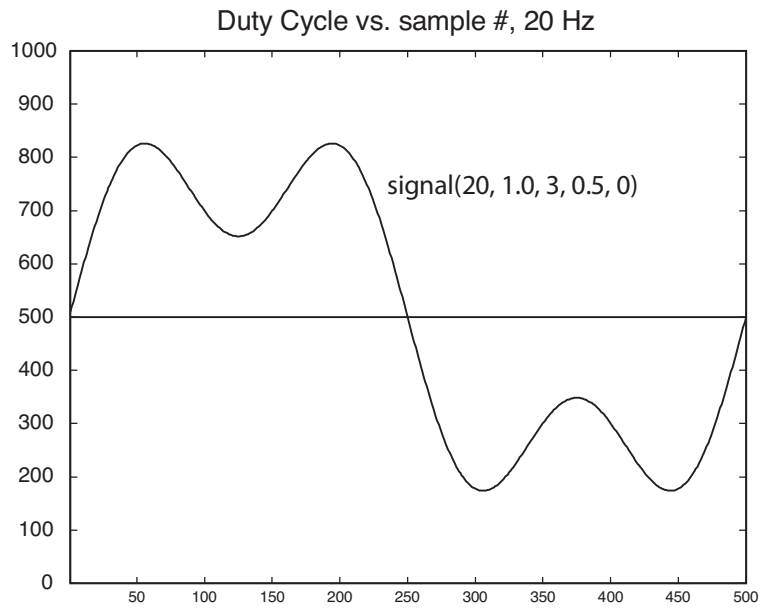


Figure 9.5: An example analog output waveform from Problem 2, plotted as the duration 0 to 1000 of the high portion of the PWM waveform, which has a period of 1000.

where the constant C is 1.65 V (half of the full range 0 to 3.3 V), A_1 is the amplitude of the sinusoid at frequency f, A_2 is the amplitude of the sinusoid at frequency kf, and ϕ is the phase offset of the higher frequency component. Typically values of A_1 and A_2 would be 1 V or less so the analog output is not saturated at 0 or 3.3 V. The function takes A_1, A_2, k, f, and ϕ as input and creates an array `dutyvec`, of appropriate length, where each entry is a value 0 to 1000 corresponding to the voltage range 0 to 3.3 V. Each entry of `dutyvec` corresponds to a time increment of 1/10 kHz = 0.1 ms, and `dutyvec` holds exactly one cycle of the analog waveform, meaning that it has $n = 10$ kHz$/f$ elements. A Matlab implementation is given below. You can experiment plotting waveforms or just use the function for reference. A reasonable call of the function is `signal(20, 0.5, 2, 0.25, 45)`, where the phase 45 is in degrees. An example waveform is shown in Figure 9.5.

```
function signal(BASEFREQ,BASEAMP,HARMONIC,HARMAMP,PHASE)

% This function calculates the sum of two sinusoids of different
% frequencies and populates an array with the values.  The function
% takes the arguments
%
% * BASEFREQ:  the frequency of the low frequency component (Hz)
% * BASEAMP:   the amplitude of the low frequency component (volts)
% * HARMONIC:  the other sinusoid is at HARMONIC*BASEFREQ Hz; must be
%              an integer value > 1
% * HARMAMP:   the amplitude of the other sinusoid (volts)
% * PHASE:     the phase of the second sinusoid relative to
%              base sinusoid (degrees)
%
% Example matlab call:  signal(20,1,2,0.5,45);

% some constants:

MAXSAMPS = 1000;     % no more than MAXSAMPS samples of the signal
ISRFREQ = 10000;     % frequency of the ISR setting the duty cycle; 10kHz
```

```
% Now calculate the number of samples in your representation of the
% signal; better be less than MAXSAMPS!

numsamps = ISRFREQ/BASEFREQ;
if (numsamps>MAXSAMPS)
  disp('Warning: too many samples needed; choose a higher base freq.');
  disp('Continuing anyway.');
end
numsamps = min(MAXSAMPS,numsamps);  % continue anyway

ct_to_samp = 2*pi/numsamps;         % convert counter to time
offset = 2*pi*(PHASE/360);          % convert phase offset to signal counts

for i=1:numsamps  % in C, we should go from 0 to NUMSAMPS-1
  ampvec(i) = BASEAMP*sin(i*ct_to_samp) + ...
              HARMAMP*sin(HARMONIC*i*ct_to_samp + offset);
  dutyvec(i) = 500 + 500*ampvec(i)/1.65; % duty cycle values,
                                         % 500 = 1.65 V is middle of 3.3V
                                         % output range
  if (dutyvec(i)>1000) dutyvec(i)=1000;
  end
  if (dutyvec(i)<0) dutyvec(i)=0;
  end
end

% ampvec is in volts; dutyvec values are in range 0...1000

plot(dutyvec);
hold on;
plot([1 1000],[500 500]);
axis([1 numsamps 0 1000]);
title(['Duty Cycle vs. sample #, ',int2str(BASEFREQ),' Hz']);
hold off;
```

(b) Write a function using the NU32 library that prompts the user for A_1, A_2, k, f, and ϕ. The array `dutyvec` is then updated based on the input.

(c) Use Timer2 and OC1 to create a PWM signal at 10 kHz. Enable the Timer2 interrupt, which generates an IRQ at every Timer2 rollover (10 kHz). The ISR for Timer2 should update the PWM duty cycle with the next entry in the `dutyvec` array. When the last element of the `dutyvec` array is reached, wrap around to the beginning of `dutyvec`. Use the shadow register set for the ISR.

(d) Choose reasonable values for RC for your RC filter. Justify your choice.

(e) The main function of your program should sit in an infinite loop, asking the user for new parameters. In the meantime, the old waveform continues to be "played" by the PWM. For the values given in Figure 9.5, use your oscilloscope to confirm that your analog waveform looks correct.

Your code will be graded on organization, comments, simplicity/elegance, and correctness. Turn in your C file for testing.

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). 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 3.3 V/1024 = 3 mV resolution. Typically used with sensors that produce analog voltages, the ADC can take nearly one million readings per second.

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 Fig 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}. By selecting both the + and − inputs to the differencing amp, we control not only the pin that is sampled but also the voltage range that it is compared against.

The differencing amp subtracts the two input voltages, sending the difference ($V_{SHA} = V_{INH} - V_{INL}$) 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.

The Successive Approximation Register (SAR) converts V_{SHA} to a 10-bit result, whose value depends on the low (V_{REFL}) and high (V_{REFH}) reference voltages. If $V_{SHA} = V_{REFH}$, the 10-bit result is $2^{10} - 1 = 1023$. For a voltage V_{SHA} that is x% of the way from V_{REFL} to V_{REFH}, the 10-bit result is 1023 × x/100. (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 don't 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

[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}.

[2] A word of caution: the ADC1BUFx buffers are not contiguous in memory: each buffer is 4-bytes long, but they are 16 bytes apart.

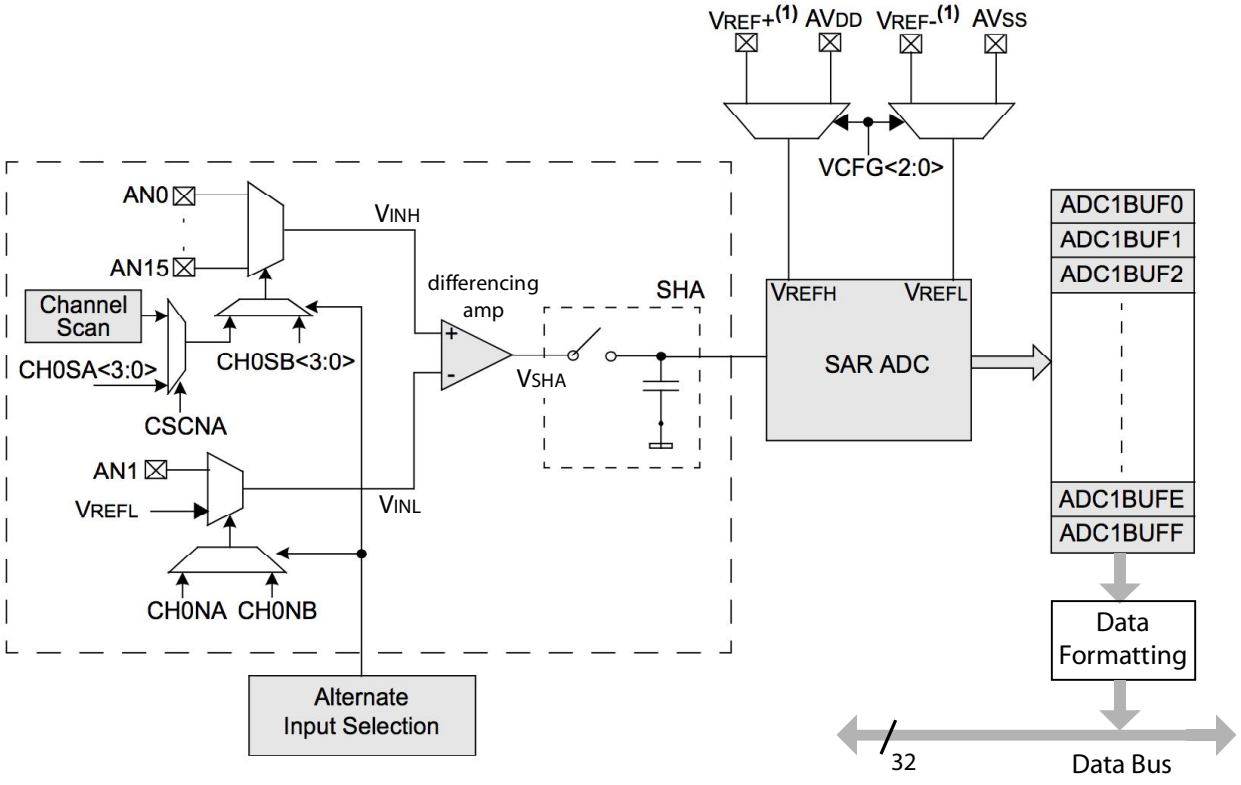


Figure 10.1: A simplified schematic of the ADC module.

Sheet, this time is 132 ns when the SAR ADC uses the external voltage references V_{REF-} and 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 this 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 microsecond. 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-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, before returning to the first address in ADC1BUF. The number of samples taken before starting after a specified number of conversions.
- 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 V − 1 V)/1024 = 1 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 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:

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 default on reset is all zeroes (analog inputs). The analog input pins AN0-AN15 correspond to the port B pins RB0-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.

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 both the Vr+ and Vr− inputs to the SAR. These references determine what voltage a given reading corresponds to: 0x000 corresponds to Vr− and 0x3FF corresponds to Vr+.

0b000 Use the internal references: Vr+ is 3.3 V and Vr− is 0 V.

0b001 Use an external reference for Vr+ and an internal reference for Vr−: Vr+ is the voltage on the Vref+ pin and Vr− is 0 V.

0b010 Use an internal reference for Vr+ and an external reference for Vr−: Vr+ is 3.3 V and Vr− is the voltage on the Vref− pin.

0b000 Use external references: Vr+ is the voltage on the Vref+ pin and Vr− is the voltage on the Vref− 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 split buffer mode (AD1CON2bits.BUFM = 1). Determines which buffer the ADC is currently filling.

1 The ADC is filling buffers 0x8-0xF, so the user should read from buffers 0x0-0x7

0 The ADC is filling buffers 0x0-0x7, so the user should read from buffers 0x8-0xF

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 - ADC1BUF7 and ADC1BUF8 - 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 - ADC1BUFF.

AD1CON3 Controls settings for the ADC clock 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

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

Tad must be at least 65 ns, so with an 80 Mhz peripheral bus clock frequency, the minimum value for this field is 3, which yields a 75ns Tad.

AD1CHS This SFR determines the which pins will be sampled (the "positive" inputs) and what they will be compared to (i.e., V_{REF-} 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 Vr- (default). Vr- 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⟩).

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 microseconds total. In this program we set the ADC clock period Tad to be $6 \times$ 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 were ready.

Code Sample 10.1. `ADC_Read2.c` Reading two analog inputs with manual initialization of sampling initialization and conversion.

```
//#define NU32_STANDALONE  // uncomment if program is standalone, not bootloaded
#include "NU32.h"          // config bits, constants, funcs 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 and 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 pin AN14 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, min flash wait, 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_WriteUART1(msg);
    _CP0_SET_COUNT(0);                    // delay to prevent a flood of messages
    while(_CP0_GET_COUNT() < DELAY_TICKS) {
      ;
    }
  }
  return 0;
}
```

10.3.2 Maximum Possible Sample Rate

The program `ADC_max_rate.c` reads from a single analog input, AN0, 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 8 samples and then generate an interrupt. The ISR reads 8 samples from ADCBUF0-7 or ADCBUF8-F while the ADC fills the other 8-word section. The ISR must finish reading one 8-word section before the other 8-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-} (RA9) to be connected to ground and pin V_{REF+} (RA10) 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, but, for this particular example, we can omit those details.

[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.

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 nine seconds).

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, AN0, at the maximum speed
// that fits the PIC32 Electrical Characteristics and the 80 MHz PBCLK
// (Tpb = 12.5 ns).  The input to AN0 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.
//
// RA9/VREF- must be connected to ground and RA10/VREF+ connected to 3.3 V.
//

//#define NU32_STANDALONE  // uncomment if program is standalone, not bootloaded
#include "NU32.h"          // config bits, constants, funcs 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)

// 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, IPL7SRS) 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 now 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)
    LATAINV = 0x20;
    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.PCFG0 = 1;          //             make RB0/AN0 an analog input
  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 = 7;             // INT step 4: IPL7, 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_WriteUART1(msg);
  NU32_WriteUART1("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_WriteUART1(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_WriteUART1(msg);
    prev_time = isr_time[i];
  }
  NU32_WriteUART1("\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)
```

and show 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 samples is $150\ ns + (12 * 75\ ns) = 1050ns$, 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 .5 Tad. In data sheets "typically" is no guarantee: in this particular case the Data Sheet states that the typical values

were not tested. Our experiment indicates that the typical times are actually a little lower than their 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 to 3.3 V in just over 2 microseconds. 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 long section in the Reference Manual.

10.5 Exercises

Chapter 11

UART

The universal asynchronous receiver/transmitter (UART) allows two devices to communicate with each other. Formerly ubiquitous as the hardware powering serial ports, the UART has been almost completely replaced by the universal serial bus (USB). Although obsolete to the average computer user, UARTs remain important in embedded programming due to their relative simplicity. You have used the PIC32's UART to communicate with your computer. Rather than connect to a physical serial port, software on your computer creates a virtual serial port that communicates with a chip (the FTDI FT2232HL) via USB. That chip converts the data it receives via USB into signals appropriate for a UART: the PIC32 does not know the difference.

11.1 Overview

The PIC32 has six UARTs, including one that the NU32 uses for bootloader communication and another that the NU32 uses for general communication between your computer and the PIC32. UARTs require a minimum of two pins: one for receiving (RX) and one for transmitting (TX). Both devices operating on the UART must also share a common ground reference (GND). The UART can simultaneously send and receive data, a feature known as full duplex communication. For one-way communication only one wire is required. The PIC32 uses the UART to communicate with data terminal equipment (DTE), a device such as your computer or another PIC32.[1]

The important parameters for basic UART communication are the baud, the data length, the parity, and the number of stop bits. The baud dictates the speed at which bits are transmitted, in bits per second. The UART sends bits in groups of either 8 or 9 bits, depending on the data length. For data lengths of 8 bits, an additional parity bit can be used to detect transmission errors. Finally, the number of stop bits indicates the minimum time between consecutive transmissions; devices can use either one or two stop bits. Both receiver and transmitter must use the same parameters to communicate. Oscillator imperfections cause the actual and nominal baud for each device to differ; thus, the UART can tolerate slight differences in baud between devices. The UART section of the Reference Manual provides information about baud tolerances.

Fig 11.1 shows a typical UART transmission. When not transmitting, the TX line is high. To start a transmission, the UART lowers TX for one baud period. This start bit tells the receiver that a transmission has begun so that the receiver can start its baud clock and begin receiving bits. Next, the data bits are sent: each bit is held on the line for one baud period. The bits are sent LSB first (e.g., the first bit sent for 0b11001000 will be a zero). Following the data bits, a parity bit may be optionally sent. Parity can be either even or odd; the transmitter adjusts the parity bit so that the total number of ones sent matches the parity setting. If the receiver expects odd (even) parity but receives an even (odd) number of ones, it knows a bit was flipped during transmission. Finally, the transmitter holds the line high, transmitting each of the stop bits. After the stop bits have been transmitted, another transmission may begin; thus using two stop bits provides the devices with extra processing time between transmissions. Notice that the baud determines how long each signal is held on the line, prior to a change being allowed. Some bits, such as the start bit,

[1] We use the term DTE loosely, to distinguish between the PIC32 and the device with which it communicates. Technically, the PIC32 is also DTE.

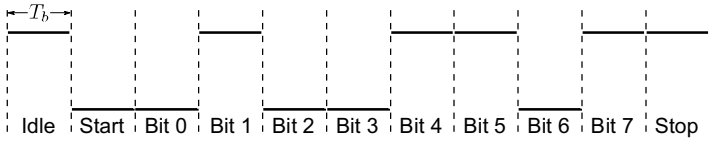


Figure 11.1: UART transmission of 0b10110010 with no parity and one stop bit.

parity bit, and stop bits, are control bits: they do not contain data. Therefore, the baud does not directly correspond to the data rate.

As the UART receives data, hardware shifts each bit into a register. When a full byte has been received, that byte is transferred into the UART's RX first-in first-out queue (FIFO). When transmitting data, software loads bytes into the TX FIFO. The hardware then loads bytes from the FIFO into a shift register, which sends them over the wire. If either FIFO is full and another byte needs to be added an overrun condition occurs and the data is lost. Preventing a TX FIFO overrun means that the software does not attempt to write to the UART unless the TX FIFO has space. To prevent an RX FIFO overrun, software must read the RX FIFO fast enough so that it always has space when received data arrives. Hardware maintains flags indicating the status of the FIFOs and can also interrupt based on the number of items in the FIFOs.

A feature called hardware flow control can help software prevent overruns. Hardware flow control requires two additional wires: request to send ($\overline{\text{RTS}}$) and clear to send ($\overline{\text{CTS}}$). Whenever the RX FIFO is full, the UART hardware de-asserts (drives high) $\overline{\text{RTS}}$, which tells the DTE not to send data. Whenever the RX FIFO has space available, the hardware asserts (drives low) $\overline{\text{RTS}}$, allowing the DTE to send data. The DTE controls $\overline{\text{CTS}}$. Whenever the DTE de-asserts $\overline{\text{CTS}}$ the PIC32 will not transmit data. For hardware flow control to work, both the DTE and PIC32 must respect the flow control signals. By default, when you use `make screen` or `make putty`, those terminal emulators configure your DTE to use hardware flow control.

Other mechanisms for flow control exist. One method, simplex mode also uses two hardware signals, but interprets them differently. Software flow control uses reserved bytes that allow each device to indicate their readiness to receive data. An escape character allows the reserved bytes to be treated as regular data.

We have outlined the basics of UART operation. Many other options exist; far too many to cover here. The guiding principle behind all UART operation, however, remains the same: both ends of the communication must agree on all the options. When interfacing with specific devices, read the data sheet and select the appropriate options.

11.2 Details

We now provide you with a description of the UART registers. The "x" in the SFR names stands for a number from 1-6.

UxMODE Enables or disables the UART. Determines the parity, number of data bits, number of stop bits, and flow control method.

UxMODE⟨15⟩ or UxMODEbits.ON: when set to one, enables the UART

UxMODE⟨9:8⟩ or UxMODEbits.UEN: Determines which pins the UART uses. Common choices are

0b00 Only the UxTX and UxRX are used (the minimum required for UART communication)

0b10 UxTX, UxRX, UxCTS, and UxRTS are used. This enables hardware flow control.

UxMODE⟨3⟩ or UxMODEbits.BRGH: A multiplier used for determining the baud, 1 = 4x multiplier (high speed), 0 = 16x multiplier (standard). Typically, we use the standard speed.

UxMODE⟨2:1⟩ or UxMODEbits.PDSEL: Determines the parity and number of data bits. Typically we use 0b00 8-bit data, no parity

UxMODE⟨0⟩ or UxMODEbits.STSEL: The number of stop bits. 0 = 1 stop bit, 1 = 2 stop bits.

UxSTA Contains the status of the UART: error flags and busy status. Controls the conditions under which interrupts occur. Also allows the user to turn the transmitter or receiver on and off.

UxSTA⟨15:14⟩ or UxSTAbits.UTXISEL: Determines when to generate a TX interrupt. The PIC32 can hold 8 bytes in its TX FIFO. Interrupts will continue to happen until the condition causing the interrupt ends.

- 0b10 Interrupt while the transmit buffer is empty.
- 0b01 Interrupt after everything in the FIFO has been transmitted.
- 0b00 Interrupt whenever the FIFO is not full.

UxSTA⟨12⟩ or UxSTAbits.URXEN: When set, enables the UART's RX pin.

UxSTA⟨10⟩ or UxSTAbits.UTXEN: When set, enables the UART's TX pin.

UxSTA⟨9⟩ or UxSTAbits.UTXBF: When set, indicates that the transmit buffer is full. If you attempt to write to the UART when the buffer is full the data will be ignored.

UxSTA⟨8⟩ or UxSTAbits.TRMT: When clear, indicates that there is no pending transmission or data in the TX buffer.

UxSTA⟨7:6⟩ or UxSTAbits.URXISEL: Determines when UART receive interrupts are generated. The PIC32 can hold 8 bytes in its RX FIFO. The interrupt will continue to happen until the condition causing the interrupt is cleared.

- 0b10 Interrupt whenever the receive buffer contains six or more characters.
- 0b01 Interrupt whenever the receive buffer contains four or more characters.
- 0b00 Interrupt whenever the receive buffer contains at least one character.

UxSTA⟨3⟩ or UxSTAbits.PERR: Set when the parity of the received data is incorrect. For even (odd) parity the UART expects the total number of received ones (including the parity bit) to be even (odd). If not using a parity bit (the usual case in this book), then there can be no parity error, but you also lose the data integrity check that parity provides.

UxSTA⟨2⟩ or UxSTAbits.FERR: Set when a framing error occurs. A framing error happens when the UART does not detect the stop bit. This often occurs if there is a baud mismatch.

UxSTA⟨1⟩ or UxSTAbits.OERR: Set when the receive buffer is full but the UART is sent another byte. When this bit is set the UART cannot receive data; therefore, if an overrun occurs you must manually clear this bit to continue receiving data. Clearing this bit flushes the data in the receive buffer, so you may want to read the bytes in the receive buffer prior to clearing.

UxSTA⟨0⟩ or UxSTAbits.URXDA: When set, indicates that the receive buffer contains data.

UxTXREG Use this SFR to transmit data. Writing to UxTXREG places the data in an 8 byte long hardware FIFO. The transmitter removes data from the FIFO and loads it into an internal shift register, UxTSR, where the data is shifted out onto the TX line, bit by bit. Once done shifting, hardware removes the next byte and begins transmitting it.

UxRXREG Hardware shifts data bit by bit into an internal RX shift register. After receiving a full byte, hardware transfers it from into the RX FIFO. Reading from UxRXREG removes a byte from the RX FIFO. If you do not read from UxRXREG often enough, the hardware queue may overrun. If the queue is full, subsequent received bytes are discarded and an overrun error status flag is set.

UxBRG Controls the baud. The value for this register can be derived from the baud as follows:

$$UxBRG = \frac{F_{PB}}{M \times B} - 1 \tag{11.1}$$

where F_{PB} is the peripheral bus frequency, $M = 4$ if UxMODE.BRGH = 1, $M = 16$ if UxMODE.BRGH = 0, and B is the desired baud.

11.3 Sample Code

11.3.1 Loopback

In our first example, the PIC32 uses UART3 to talk to itself. Connect U3RX (RG7) to U3TX (RG8). We also set the baud to an extremely low rate (100) so that you can easily view the transactions on an oscilloscope or logic analyzer. If you set the oscilloscope into single capture mode and trigger on the falling edge, you should be able to see the full transaction. We send two bytes on the UART so that you can verify the stop bits. The baud is 100, so changes on the line occur at 100 Hz.

Code Sample 11.1. UART code that talks to itself.

```
//#define NU32_STANDALONE  // uncomment if program is standalone, not bootloaded
#include "NU32.h"          // config bits, constants, funcs for startup and UART

// We will setup UART3 at a slow baud rate so you can
// easily examine it on your scope
// Connect the UART3 RX and TX pins together so that the UART can communicate with itself.
int main(void) {
  char msg[100] = {};
  NU32_Startup();
  // now initialize UART3, 76 baud, odd parity, 1 stop bit

  U3MODEbits.PDSEL = 0x2; // odd parity (parity bit set to make the number of 1's odd
  U3STAbits.UTXEN = 1;    // enable transmit
  U3STAbits.URXEN = 1;    // enable receive

  // baud = Fpb/(16*(U3BRG+1))
  // U3BRG = Fpb/(16*baud) -1
  // setup for 100 baud.  This means 100 bits /sec or 1 bit/ 1/10ms
  U3BRG = 49999;
  U3MODEbits.ON = 1; // turn on the uart

  //scope instructions:
  // scale 1 division is 10ms
  // trigger on falling edge
  // single capture
  // then, after you send the byte, the edge will fall, and you can see the transmission on the scope
  // maybe can use digital in on nuscope>?
  while(1) {
    unsigned char data = 0;
    NU32_WriteUART1("Enter hex byte (lowercase) to send to UART3 (i.e., 0xa1): ");
    NU32_ReadUART1(msg,100);
    sscanf(msg,"%2x",&data);
    sprintf(msg,"0x%02x\r\n",data);
    NU32_WriteUART1(msg); //echo back

    // 1st wait for uart to be ready to transmit
    while(U3STAbits.UTXBF) {
      ;
    }
    //write twice so we can see the stop bit on a scope. We know the fifo has 8 bytes so no need to check if there
    U3TXREG = data;
    U3TXREG = data;
    //wait to receive
    while(!U3STAbits.URXDA) {
      ;
    }
```

```
    // read the byte
    data = U3RXREG;
    while(!U3STAbits.URXDA) { // read the data again since we wrote two bytes
      ;
    }
    data = U3RXREG;
    sprintf(msg,"Read 0x%x from UART3\r\n",data);
    NU32_WriteUART1(msg);
  }
  return 0;
}
```

11.3.2 Interrupt Based

The next example demonstrates the use of interrupts. Interrupts can be generated based on the number of elements in the RX or TX buffers, or whether an error has occurred. For example, you can interrupt whenever the RX buffer is half full or when the TX buffer is empty. The IRQs for these interrupts share the same vector; therefore, you must check within the ISR to see what event triggered it. You must also remove the condition that triggered the interrupt or it will trigger again after you exit the ISR. The code below triggers an interrupt when the RX buffer contains at least one character. It reads the received value and transmits it immediately. Using interrupts for handling serial I/O allows the PIC32 to perform tasks and still receive data from the serial port, even without explicitly checking for it.

Code Sample 11.2. UART code that uses interrupts to receive data.

```
#include "NU32.h"

#define DAT_SIZE 100

void __ISR(_UART_1_VECTOR, IPL1SOFT) IntUart1Handler(void) {
  if (IFS0bits.U1RXIF) {
    U1TXREG = U1RXREG; // send the typed data out
    IFS0bits.U1RXIF = 0;
  } else if(IFS0bits.U1TXIF) {
    // this is a tx interrupt. we did nto enable tx interrupts, so do nothing
  } else {
    // this is an error interrupt. you can check U1STA for the specific error flag.
  }
}

int main(void) {
  NU32_Startup();
  NU32_LED1 = 1;
  NU32_LED2 = 1;
  __builtin_disable_interrupts();
  U1MODEbits.ON = 0;
  _nop();   // wait one instruction before modifying U1 registers, as per data sheet

  U1MODEbits.BRGH = 0; // set baud to 230400
  U1BRG = ((NU32_SYS_FREQ / 230400) / 16) - 1;

  // configure TX & RX pins as output & input pins
  U1STAbits.UTXEN = 1;
  U1STAbits.URXEN = 1;

  // configure using RTS and CTS
```

```
  U1MODEbits.UEN = 2;

  // configure the uart interrupts
  U1STAbits.URXISEL = 0x0; // rx interrupt when receive buffer not empty (at least 1 byte has been received)

  IFS0bits.U1RXIF = 0;
  IFS0bits.U1TXIF = 0;
  IPC6bits.U1IP = 1;      //interrupt priority
  IEC0bits.U1RXIE = 1;    // enable the rx interrupt
  // enable the uart
  U1MODEbits.ON = 1;
  __builtin_enable_interrupts();
  while(1) {
    _nop();
  }

  return 0;
}
```

11.3.3 NU32 Library

The NU32 library contains three functions that access the UART: `NU32_Setup`, `NU32_ReadUART1` and `NU32_WriteUART1`. The setup code configures UART1 for a baud of 230400, one stop bit, 8 data bits, no parity bit, and hardware flow control. No UART interrupts are used. Notice that `NU32_ReadUART1` keeps reading from the UART until it receives a certain control character (`'\n'` or `'\r'`); thus it will wait indefinitely for input before proceeding. The function `NU32_WriteUART1` waits for empty space in the TX buffer before sending data. Thus, if the computer holds $\overline{\text{CTS}}$ high, then `NU32_WriteUART1` will wait until the computer is ready to send data. If the computer's serial port has flow control enabled, then the PIC32 will wait to send data until you open the port in a terminal emulator or some other application. Unfortunately, the default flow control settings for the serial port depends on your operating system; thus, flow control may not be enabled until you open the port in your terminal emulator.

Code Sample 11.3. NU32 Library

```
#include "NU32.h"

#define NU32_DESIRED_BAUD 230400 // Baudrate for RS232


// Perform startup routines:
//  Make NU32_LED1 and NU32_LED2 pins outputs (NU32USER is by default an input)
//  Initialize the serial port - UART1 (no interrupt)
//  Enable interrupts
void NU32_Startup() {
  // disable interrupts
  __builtin_disable_interrupts();

  // enable the cache (if used in STANDALONE mode, this is necessary)
  // This command sets the CP0 CONFIG register
  // the lower 4 bits can be either 0b0011 (0x3) or 0b0010 (0x2)
  // to indicate that kseg0 is cacheable (0x3) or uncacheable (0x2)
  // see Chapter 2 "CPU for Devices with M4K Core" of the PIC32 reference manual
  // most of the other bits have prescribed values
  // microchip does not provide a _CP0_SET_CONFIG macro, so we directly use
  // the compiler built-in command _mtc0
```

```
  __builtin_mtc0(_CP0_CONFIG, _CP0_CONFIG_SELECT, 0xa4210583); // 0xa4210582 would  disable cache

  // set the prefectch cache wait state to 2, as per the
  // electrical characteristics data sheet
  CHECONbits.PFMWS = 0x2;

  //enable prefetch for cacheable and noncacheable memory
  CHECONbits.PREFEN = 0x3;

  // 0 data RAM access wait states
  BMXCONbits.BMXWSDRM = 0x0;

  // enable multi vector interrupts
  INTCONbits.MVEC = 0x1;

  // disable JTAG to get A4 and A5 back
  DDPCONbits.JTAGEN = 0;

  TRISACLR = 0x0030; // Make A5 and A4 outputs (LED2 and LED1)
  NU32_LED1 = 1;      // LED1 is off
  NU32_LED2 = 0;      // LED2 is on

  // turn on UART1 without an interrupt
  U1MODEbits.BRGH = 0; // set baud to NU32_DESIRED_BAUD
  U1BRG = ((NU32_SYS_FREQ / NU32_DESIRED_BAUD) / 16) - 1;

  // 8 bit, no parity bit, and 1 stop bit (8N1 setup)
  U1MODEbits.PDSEL = 0;
  U1MODEbits.STSEL = 0;

  // configure TX & RX pins as output & input pins
  U1STAbits.UTXEN = 1;
  U1STAbits.URXEN = 1;
  // configure using RTS and CTS
  U1MODEbits.UEN = 2;

  // enable the uart
  U1MODEbits.ON = 1;

  __builtin_enable_interrupts();
}

// Read from UART1
// block other functions until you get a '\r' or '\n'
// send the pointer to your char array and the number of elements in the array
void NU32_ReadUART1(char * message, int maxLength) {
  char data = 0;
  int complete = 0, num_bytes = 0;
  // loop until you get a '\r' or '\n'
  while (!complete) {
    if (U1STAbits.URXDA) { // if data is available
      data = U1RXREG;      // read the data
      if ((data == '\n') || (data == '\r')) {
        complete = 1;
      } else {
        message[num_bytes] = data;
        ++num_bytes;
        // roll over if the array is too small
        if (num_bytes >= maxLength) {
```

```
          num_bytes = 0;
        }
      }
    }
  }
  // end the string
  message[num_bytes] = '\0';
}

// Write a charater array using UART1
void NU32_WriteUART1(const char * string) {
  while (*string != '\0') {
    while (U1STAbits.UTXBF) {
      ; // wait until tx buffer isn't full
    }
    U1TXREG = *string;
    ++string;
  }
}
```

11.3.4 Sending Data from an ISR

The next example demonstrates how to send data generated in a timer ISR over the UART. Writing to the UART may be too slow for the timer frequency, especially if you wish to format the data using `sprintf`. Instead, the ISR writes data to a buffer while mainline code sends it to your computer. Rather than adding data to the buffer every iteration, we only record it once every fourth time the ISR is called. This process, known as decimation, reduces the amount of RAM needed per second of data collected, at the expense of resolution.[2]

Code Sample 11.4. `batch.c` Storing data in an ISR and sending it over the UART.

```
#include "NU32.h"

#define DECIMATE 3                // only send every fourth sample (counting starts at zero)
#define NSAMPLES 5000             // store 5000 samples

volatile int data_buf[NSAMPLES]; // stores the samples
volatile int curr = 0;              // the current index into buffer

void __ISR(_TIMER_1_VECTOR, IPL5SOFT) Timer1ISR(void) {  // timer 1 isr operates at 5 kHz
  static int count = 0;   // counter used for decimation
  static int i = 0;       // the data returned from the isr
  if(count == DECIMATE) {   // skip some data
    count = 0;
    if(curr < NSAMPLES) {
      data_buf[curr] = i;    // queue a number for sending over the UART
      ++i;                 // generate the data (we just increment it for now)
      ++curr;
    }
  }
  ++count;
  IFS0bits.T1IF = 0;              // clear interrupt flag
}
```

[2] The term "decimation" comes from a disciplinary practice of the Roman Army whereby every 10th soldier would be killed.

```
void main(void) {
  int i = 0;
  char buffer[100] = {};
  NU32_Startup(); // cache on, min flash wait, interrupts on, LED/button init, UART init

  __builtin_disable_interrupts(); // INT step 2: disable interrupts at CPU
T1CONbits.TCKPS = 0b01;         // PBCLK prescaler value of 1:8
PR1 = 1999;          // The frequency is 80 MHz / (8 * (1999 + 1)) = 5 kHz
TMR1 = 0;
IPC1bits.T1IP = 5;    // interrupt priority 7
IFS0bits.T1IF = 0;    // clear the interrupt flag
IEC0bits.T1IE = 1;    // enable the interrupt
T1CONbits.ON  = 1;    // turn the timer on                                  // INT step 3: setup peripheral
  __builtin_enable_interrupts();  // INT step 7: enable interrupts at CPU

  NU32_ReadUART1(buffer,100);  // wait for the user to press enter before continuing
  while(curr !=NSAMPLES) { ; } // wait for the data to be collected
    sprintf(buffer,"%d\r\n",NSAMPLES); // send the number of samples
    NU32_WriteUART1(buffer);
    for(i = 0; i < NSAMPLES; ++i) {
      sprintf(buffer,"%d\r\n",data_buf[i]);  // send the data to the terminal
      NU32_WriteUART1(buffer);
   }
  while(1) { _nop(); }
}
```

The above example waits for the buffer to be full. It then sends the buffer's length, followed by each buffer item over the UART. Although relatively simple, this method has its drawbacks. By writing all the data to a buffer and then sending it over the UART we introduce a potentially long delay between when the data is generated and when it is sent. Furthermore, we must store all the data we want to send in RAM at one time.

The situation could be improved by using a circular (ring) buffer. A circular buffer is a FIFO queue implemented as an array and two index variables: `write` and `read` (see Fig. 11.2). Data is added to the array at the write location and read from the read location, after which the indexes are updated. When the indexes reach the array's end they wrap around to the beginning. Circular buffers are useful for sharing data between interrupts and mainline code. One task (in this case the interrupt) can write to the buffer while the other (in this case the mainline code) reads from the buffer; removing the need to store all data in RAM prior to sending it over the UART.

Code Sample 11.5. `circ_buf.c` Streaming data from an ISR over the UART, using a circular buffer.

```
#include "NU32.h"
// uses a circular buffer to stream data from an ISR over the UART
// notice that the buffer can be much smaller than the total number of samples sent and
// that data starts streaming immediately rather than with batch.c
#define BUFLEN   1024     // length of the buffer
#define NSAMPLES 5000     // number of samples to collect

static volatile int data_buf[BUFLEN];             // array that stores the data
static volatile unsigned int read = 0, write = 0;

static volatile int start = 0;    // set to start recording
int buffer_empty() {
  return read == write; // read and write position are the same indicating buffer is empty
}

int buffer_full() {   // return true if the buffer is full.
```

```
  return (write + 1) % BUFLEN == read;
}

int buffer_read() { // read from the buffer and store result in *val.  Assume the buffer is not empty
  int val = data_buf[read];
  ++read;
  if(read >= BUFLEN) {
    read = 0;
  }
  return val;
}

void buffer_write(int data) { // add an element to the buffer.  if the buffer is full the data is dropped
                              // note that this means one spot in the buffer will always be unused.
                              // other methods exist that do not have this disadvantage
  if(!buffer_full()) {
    data_buf[write] = data;
    ++write;
    if(write >= BUFLEN) {
      write = 0;
    }
  }
}

void __ISR(_TIMER_1_VECTOR, IPL5SOFT) Timer1ISR(void) {  // timer 1 isr operates at 5 kHz
  static int i = 0;       // the data returned from the isr
  if(start) {
    buffer_write(i);        // add the data to the buffer
    ++i;                    // modify the data (here we just increment it as an example)
  }
  IFS0bits.T1IF = 0;      // clear interrupt flag
}

int main(void) {
  int sent = 0;
  char msg[100] = {};
  NU32_Startup(); // cache on, min flash wait, interrupts on, LED/button init, UART init

  __builtin_disable_interrupts(); // INT step 2: disable interrupts at CPU
T1CONbits.TCKPS = 0b01;         // PBCLK prescaler value of 1:8
PR1 = 1999;          // The frequency is 80 MHz / (8 * (1999 + 1)) = 5 kHz
TMR1 = 0;
IPC1bits.T1IP = 5;    // interrupt priority 7
IFS0bits.T1IF = 0;    // clear the interrupt flag
IEC0bits.T1IE = 1;    // enable the interrupt
T1CONbits.ON  = 1;    // turn the timer on                                  // INT step 3: setup peripheral
  __builtin_enable_interrupts();  // INT step 7: enable interrupts at CPU

  NU32_ReadUART1(msg,100); // wait for the user to press enter before continuing
  sprintf(msg, "%d\r\n", NSAMPLES); // tell the client how many samples to expect
  NU32_WriteUART1(msg);
  start = 1;
  for(sent = 0; sent < NSAMPLES; ++sent) { // send the samples to the client
   while(buffer_empty()) { ; }    // wait for data to be in the queue
    sprintf(msg,"%d\r\n", buffer_read()); // read from the buffer and send the data over uart
    NU32_WriteUART1(msg);
  }
  while(1) { _nop(); }
  return 0;
```

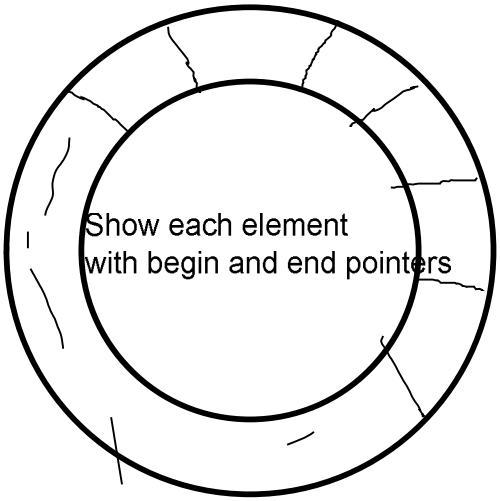


Figure 11.2: A circular buffer.

```
}
```

Notice that if the buffer is full, data is skipped. You can experiment with this setting by changing the size of the buffer and the number of samples requested. If the buffer is too small, it will become full, and you will see numbers being skipped in the output. As with all code that shares data between interrupts and mainline code, the relevant variables are declared `volatile`. Also, since the mainline code can be interrupted at any point, the ordering of operations is important. For instance, updating the read index prior to reading the data could cause the ISR to overwrite the data. Also, there are more efficient implementations of the circular buffer; however, we sacrificed performance for clarity.

11.3.5 Communication With MATLAB

So far, when we have used the UART to communicate with your computer, you have opened the serial port in a terminal emulator. MATLAB can also open serial ports, allowing you to automate communication with the PIC32 and plot data from it. As a first example, we will communicate with `talkingPIC.c` from MATLAB. First, load `talkingPIC.c` onto the PIC32 (see Ch. **??** for the code).

Next, open MATLAB and edit `talkingPIC.m`. You will need to edit the first line and set the port to be the `PORTA` value from your `Makefile`.

Code Sample 11.6. `talkingPIC.m` Simple Matlab code to talk to `talkingPIC` on the PIC32.

```
port='COM3'; % Edit this with the correct name of your PORTA.

% Makes sure port is closed
if ~isempty(instrfind)
    fclose(instrfind);
    delete(instrfind);
end
fprintf('Opening port %s....\n',port);

% Defining serial variable
mySerial = serial(port, 'BaudRate', 230400, 'FlowControl', 'hardware');

% Opening serial connection
```

```
fopen(mySerial);

% Writing some data to the serial port
fprintf(mySerial,'%f %d %d\n',[1.0,1,2])

% Reading the echo from the PIC32 to verify correct communication
data_read = fscanf(mySerial,'%f %d %d')

% Closing serial connection
fclose(mySerial)
```

The code, `talkingPIC.m`, opens a serial port, sends three numerical values to the PIC32, receives the values, and closes the port. Run `talkingPIC.c` on your PIC32, then execute `talkingPIC.m` in Matlab.

We can also combine Matlab with `batch.c` or `circ_buf.c`, allowing us to plot data received from an ISR. The example below reads the data produced by `streaming.c` and plots it in Matlab. Once again, change the `port` variable to match the serial port that your PIC32 uses.

Code Sample 11.7. `uart_plot.m` Matlab code to plot data received from the UART.

```
% plot streaming data in matlab
port ='/dev/ttyUSB0'

if ~isempty(instrfind)  % closes the port if it was open
  fclose(instrfind);
  delete(instrfind);
end

mySerial = serial(port, 'BaudRate', 230400, 'FlowControl','hardware');
fopen(mySerial);

fprintf(mySerial,'%s','\n'); %send a newline to tell the PIC32 to send data

len = fscanf(mySerial,'%d'); % get the length of the matrix

data = zeros(len,1);

for i = 1:len
  data(i) = fscanf(mySerial,'%d'); % read each item
end

plot(1:len,data);                % plot the data
```

11.3.6 Communication With Python

You can also communicate with the PIC32 from the Python programming language. This freely available scripting language has many libraries available that help it be used as an alternative to MATLAB. To communicate over the serial port you need the `pyserial` library. For plotting, we use the libraries `matplotlib` and `numpy`. The following code reads data from the PIC32 and plots it. As with the Matlab code, you need to specify your own port where the `port` variable defined. This code will plot data from generated from either `batch.c` or `circ_buf.c`.

Code Sample 11.8. `uart_plot.py` Python code plot data received from the UART.

```
#!/usr/bin/python
# Plot data from the PIC32 in python
```

```
# requries pyserial, matplotlib, and numpy
import serial
import matplotlib.pyplot as plt
import numpy as np

port = '/dev/ttyUSB0' # the name of the serial port



with serial.Serial(port,230400,rtscts=1) as ser:
  ser.write("\n".encode())     #signal to the pic to send data. must call encode to convert to byte array
  line = ser.readline()
  nsamples = int(line)
  x = np.arange(0,nsamples) # x is [1,2,3,... nsamples]
  y = np.zeros(nsamples)# x is 1 x nsamples an array of zeros and will store the data

  for i in range(nsamples): # read each sample
    line = ser.readline()   # read a line from the serial port
    y[i] = int(line)        # parse the line (in this case it is just one integer)

plt.plot(x,y)
plt.show()
```

11.4 XBee Radio

XBee radios are small, low-power radio transmitters that allow wireless communication over a long range, according to the IEEE 802.15.4 standard. XBee radios have two primary modes of operation, serial pass-through and API mode. In API mode, the PIC32 sends data over UART to the XBee, but that data must follow a specific protocol, detailed in the XBee manual [**?**]. In serial pass-through mode, both devices communicate with the XBee over UART and the XBee directly sends the data over the airwaves. In this mode you could, for instance, use the NU32 library to send data over XBee by connecting the radio to UART1.

XBee radios have numerous firmware settings that must be configured before you use them. The main setting is the wireless channel; two XBees cannot communicate unless they use the same channel. Another setting is the baud, which can be set as high as 115200. The easiest way to program an XBee is to purchase a development board, connect it to your computer, and use the X-CTU program provided by the manufacturer. Alternatively, you can program XBees directly using the API mode over a serial port (either from your computer or the PIC32).

After setting up the XBees to use the desired baud and communication channel, you can use them as a drop-in replacement for the wires connecting two UARTs. One caveat occurs at higher baud rates. The XBee generates its baud by dividing an internal clock signal. This clock cannot actually achieve a baud of 115200, rather, when set to 115200 the baud actually is 111000. Such baud rate mismatches are a common issue when using UARTs. Due to the tolerances of UART timing, the XBee may work for a time but occasionally experience failures. The solution is to set your baud to 111000 to match the actual baud of the XBee.

11.5 Chapter Summary

- A UART is the low-level engine underlying serial communication. Once ubiquitous, serial ports have been largely replaced by USB on consumer products.
- The NU32 board uses two UARTS to communicate with your PC. Software on your computer emulates a serial port, which transfers data via USB to a chip on the NU32 board. This chip then converts the

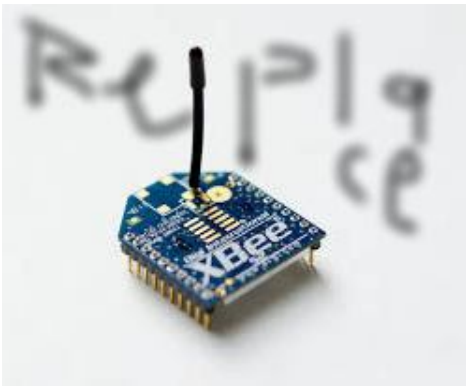


Figure 11.3: An XBee 802.15.4 Radio

USB data into a format suitable for the UART. Neither your terminal nor the PIC32 know that data is actually sent over USB, they just see a UART.

- The PIC32 maintains two 8-byte hardware FIFOs, one for receiving, one for sending. These FIFOs buffer data in hardware, allowing software to temporarily attend to other tasks.
- Both the PIC32 and the DTE must agree upon a common communication speed (baud) and data format; otherwise data will not be interpreted properly.
- Hardware flow control provides a method to signal that your device is not ready to receive more data. Although hardware handles flow control automatically, software must ensure that the RX and TX buffers do not overrun.

Chapter 12

SPI Communication

The serial peripheral interface (SPI) allows the PIC32 to communicate with other devices at high speeds. Numerous devices use SPI as their primary mode of communication, such as RAM, flash, SD cards, ADCs, DACs, and accelerometers. The main advantage of SPI is its speed and relative simplicity. The main disadvantage is the relatively high number of pins required and difficulties involved with controlling multiple devices on the same bus.

12.1 Overview

The PIC32 has four SPI peripherals; each typically requires four pins: output (SDO), input (SDI), clock (SCK), and chip select (SS) (devices also have a common ground reference). The chip select is often used directly as a digital output rather than controlled by the SPI peripheral. If only one-way communication is required the input or output can be dropped. There is also an SPI variation that allows the input and output line to be shared; however, the PIC32 peripheral does not support this mode. Figure 12.1 shows a typical SPI connection.

SPI is based upon a master-slave architecture. The SPI bus contains one master (usually the PIC32) and at least one slave. The master initiates all communication by creating a square wave on the clock line. Reading from and writing to the slave occur simultaneously; one bit per clock pulse. Transfers can occur either as 8, 16, or 32 bit operations depending on the device. The chip select pin is used to enable the slave; if multiple slaves are on the bus then the master must use multiple chip selects[1] Figure 12.2 depicts the signals for a typical SPI transaction. The polarity and timing of the clock signal may differ between devices; see Sec. 12.2 for more information.

In addition to the basic four wire SPI mode, the PIC32 also has more advanced modes of operations, which we do not cover in detail here. The PIC32 can use the SPI interface to communicate with certain audio devices. Additionally, a framing mode allows for streaming data from supported devices. By default, the SPI peripheral has only a single input and single output buffer. The PIC32 also has an enhanced buffer mode, which provide FIFOs for queuing data waiting to be transferred or processed.

12.2 Details

The SPI peripheral uses five SFRs. Many of the settings are related to the alternative operation modes that we do not discuss. SFRs and fields that we omit may be safely left at their default values for typical applications.

SPI uses a single SFR for both reading and writing data, SPIxBUF. Reads and writes happen simultaneously with SPI and only a write generates a clock signal; therefore you should read after every write to SPIxBUF.

SPIxCON This register contains the main control options for the SPI peripheral.

[1] Slave devices may also be daisy-chained, but this requires that the device supports this mode of operation.

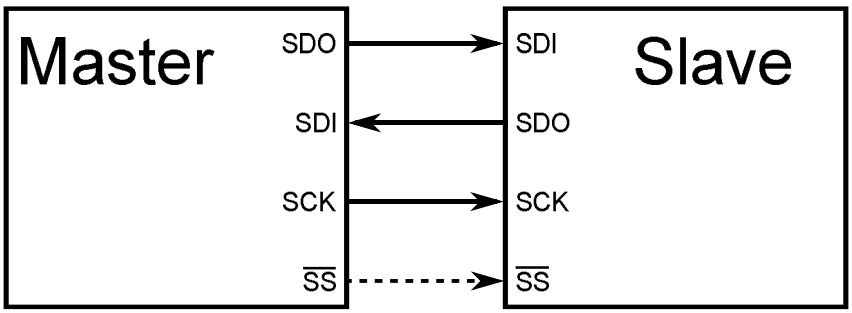
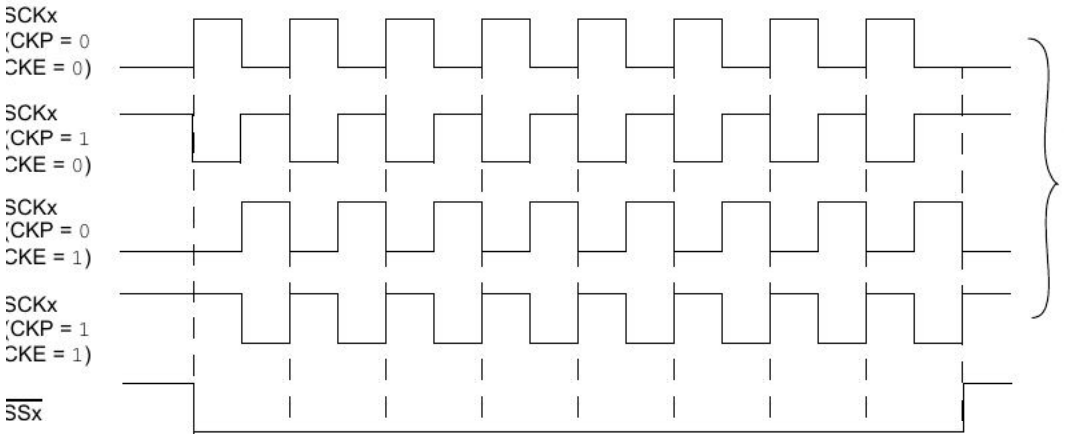


Figure 12.1: An SPI master (the PIC32) connected to a slave device. Arrows indicate data direction. The dashed arrow is an optional connection.



should drawn such that the clock is active low and the data transitions from idle to active (high to low)

Figure 12.2: Timing for an SPI transaction.

SPIxCON⟨28⟩ or SPIxCONbits.MSSEN : Master slave select enable. If set, the SPI master will assert the slave select pin prior to sending data and de-assert it after sending data. Setting this bit means the slave select pin will be controlled by the SPI peripheral rather than as a normal digital output. It should only be used if you want to toggle the chip select after every word sent to the slave. Many devices require you to toggle chip select after a complete multi-word transaction: in this case, you should have SPIxCONbits.MSSEN = 0 and use the chip select as a regular digital output.

SPIxCON⟨15⟩ or SPIxCONbits.ON : Set to enable the SPI peripheral.

SPIxCON⟨11:10⟩ or SPIxCONbits.MODE32 (bit 11) and SPIxCONbits.MODE16 (bit 10): Determines the communication width.

0b1x (SPIxCONbits.MODE32 = 1): 32 bits of data sent per transfer.

0b01 (SPIxCONbits.MODE32 = 0 and SPIxCONbits.MODE16 = 1): 16 bits of data sent per transfer.

0b00 (SPixCONbits.MODE32 = SPIxCONbits.MODE16 = 0): 8 bits of data sent per transfer.

A transfer is initiated by writing the SPI buffer.

SPIxCON⟨9⟩ or SPIxCONbits.SMP: determines when, relative to the clock pulses, the master samples the data. Should be set to match the slave device's specifications.

1 Sample at end of a clock pulse.

0 Sample in the middle of a clock pulse.

SPIxCON⟨8⟩ or SPIxCONbits.CKE: The clock signal can be configured as either active high or active low by setting SPIxCONbits.CKP (see below). This bit, SPIxCONbits.CKE, determines whether the master changes the current output bit on the edge when the clock transitions from active to idle or idle to active. You should choose this bit based on what the slave device expects. Figure 12.2 is drawn assuming that SPIxCONbits.CKE = 0.

1 The output data changes when the clock transitions from active to idle.

0 The output data changes when the clock transitions from idle to active.

SPIxCON⟨6⟩ or SPIxCONbits.CKP: The clock signal can be be configured as being active high or active low. This setting should match the expectations of the slave device. Figure 12.2 assumes that SPIxCONbits.CKP = 1.

1 The clock is idle when high, active when low.

0 The clock is idle when low, active when high.

SPIxCON⟨5⟩ or SPIxCONbits.MSTEN: Master enable. Usually the PIC32 operates as the master, meaning that it controls the clock and hence when and how fast data is transferred, in which case this bit should be set to 1. To use the SPI peripheral as a slave, this bit is cleared to 0.

1 The SPI peripheral is the master.

0 The SPI peripheral is the slave.

SPIxSTAT The status of the SPI peripheral.

SPIxSTAT⟨11⟩ or SPIxSTATbits.SPIBUSY: When set, indicates that the SPI peripheral is busy transferring data. You should not access the SPI buffer SPIxBUF when the peripheral is busy.

SPIxSTAT⟨6⟩ or SPIxSTATbits.SPIROV: Set to indicate that an overflow has occurred, which happens when the receive buffer is full and another data word is received. This bit should be cleared in software. The SPI peripheral can only receive data when this bit is clear.

SPIxSTAT⟨1⟩ or SPIxSTATbits.SPITXBF: SPI transmit buffer full. Set by hardware when you write to SPIxBUF. Cleared by hardware after the data you wrote is transferred into the transmit buffer SPIxTXB, a non-memory mapped buffer. When this bit is clear you can write to SPIxBUF.

SPIxSTAT⟨0⟩ or SPIxSTATbits.SPIRXBF: SPI receive buffer full. Set by hardware when data is received into the SPI receive buffer indicating that SPIxBUF can be read. Cleared when you read the data via SPIxBUF.

SPIxBUF Used to both read and write data over SPI. Reads and writes happen simultaneously. When, as the master, you write data to SPIxBUF, the SPI peripheral generates a clock signal and sends data over the SDOx pin. Meanwhile, in response to the clock signal, the slave sends data to the SDIx pin. To avoid buffer overflow errors, every time you write to SPIxBUF you should read from SPIxBUF, even if you do not need the data. Additionally, since the slave can only send data when it receives a clock signal, you must write prior to reading data.

SPIxBRG Determines the SPI clock frequency. Only the lower 12 bits are used. To calculate the appropriate value for SPIxBRG use the tables provided in the reference manual or the following formula:

$$SPIxBRG = \frac{F_{PB}}{2F_{sck}} - 1, \tag{12.1}$$

where F_{PB} is the peripheral bus clock frequency (80MHz for the NU32 board), and F_{sck} is the desired clock frequency. SPI can operate at relatively high frequencies, in the megahertz range. The master dictates the clock frequency and the slave reads the clock; therefore a slave device never needs to configure a clock frequency.

12.3 Sample Code

12.3.1 Loopback

The first example uses two SPI peripherals to allow the PIC32 to communicate with itself over SPI. Although practically useless, the code serves as vehicle for understanding the basic configuration of both an SPI master and an SPI slave. Both master and slave are configured to send 16 bits per transfer. Notice that the SPI master sets a clock frequency, whereas the slave does not. The program prompts the user to enter two 16-bit hexadecimal numbers. The first number is sent by the master and the second by the slave. The code then reads from the SPI ports and prints the results.

Remember, one bit of data is transferred in each direction per clock cycle. When the slave writes to the SPI buffer, the data is not actually sent until the master generates a clock signal. Both master and slave use the clock to send and receive the data. As sending and receiving happen simultaneously, the master always reads from SPIxBUF after writing to SPIxBUF.

Code Sample 12.1. `spi_loop.c` SPI loopback example.

```
#include "NU32.h"
// Demonstrates spi by using two spi peripherals on the same PIC32,
// one is the master, the other is the slave
// spi4 will be the master, spi1 the slave.
// connect
// SDO4 -> SDI1 (pin F5 -> pin C4)
// SDI4 -> SDO1 (pin F4 -> pin D0)
// SCK4 -> SCK1 (pin F13 -> pin D10)


int main(void) {
  char buf[100] = {};
  // setup NU32 LED's and buttons
  NU32_Startup();

  // Setup the master Master - SPI4, pins are: SDI4(F4), SDO4(F5), SCK4(F13)
  // since the pic is just starting, we know that spi is off. We rely on defaults here
  SPI4BUF;                   // clear the rx buffer by reading from it
  SPI4BRG = 0x4;             // baud rate to 8MHz [SPI4BRG = (80000000/(2*desired))-1]
  SPI4STATbits.SPIROV = 0;  // clear the overflow bit
  SPI4CONbits.MODE32 = 0;   // use 16 bit mode
  SPI4CONbits.MODE16 = 1;
  SPI4CONbits.MSTEN = 1;    // master operation
  SPI4CONbits.ON = 1;       // turn on spi 4

  // Slave - SPI1, pins are: SDI1(C4), SDO1(D0), SCK1(D10), SS1(D9)
  SPI1BUF;                 // clear the rx buffer
  SPI1STATbits.SPIROV = 0; // clear the overflow
  SPI1CONbits.MODE32 = 0;   // use 16 bit mode
  SPI1CONbits.MODE16 = 1;
  SPI1CONbits.MSTEN = 0;   // slave mode
  SPI1CONbits.ON = 1;      // turn spi on.  Note: in slave mode you do not
                           // set the baud (since the clock is provided by master)

  while(1) {
    unsigned short master = 0, slave = 0;
    unsigned short rmaster = 0, rslave = 0;
    NU32_WriteUART1("Enter two hex words (lowercase) to send from master and slave (i.e., 0xd13f 0xb075):\r\n");
    NU32_ReadUART1(buf,100);
    sscanf(buf,"%04hx %04hx",&master, &slave);
    // have the slave write its data to the spi buffer
```

```
    // (note, the data will not be sent until the master performs a write)
    // (normally this would happen on a different chip)
    SPI1BUF = slave;
    // now the master performs a write
    SPI4BUF = master;
    // wait until the master receives the data
    while(!SPI4STATbits.SPIRBF) {
      ; //note waiting for the master buffer to be full is sufficient
        //because data is sent synchronously so the slave simultaneously receives the data
        //in a real situation, the slave processor would be waiting for its data separately.
    }
    // receive the data
    rmaster = SPI4BUF;
    rslave = SPI1BUF;
    sprintf(buf,"Master sent 0x%04x,  Slave sent 0x%04x\r\n", master, slave);
    NU32_WriteUART1(buf);
    sprintf(buf," Slave read 0x%04x, Master read 0x%04x\r\n",rslave,rmaster);
    NU32_WriteUART1(buf);
  }
  return 0;
}
```

12.3.2 SRAM

One use for SPI is to add external RAM to the PIC32. The Microchip 23K256 static random-access memory (SRAM) has an SPI interface. The data sheet for the RAM describes its communication protocol. The SRAM has three modes of operation, byte, page, and sequential. Byte operation allows reading or writing a single byte of RAM. The RAM is divided into sections called pages. In page mode, you can access one page of RAM at a time. Finally, sequential mode allows writing or reading sequences of bytes. The example code we provide uses sequential mode exclusively, since all memory can be accessed in this mode.

The SRAM chip requires the use of a chip select pin $\overline{\text{CS}}$. This signal tells the SRAM when data or commands are being sent, and when data is finished. This signal should be asserted (brought low) prior to communicating and de-asserted (brought high) when finished. We control $\overline{\text{CS}}$ as a digital output pin, as its state is only changed after several bytes are sent, not after every byte is sent to the device (which is what the automatic slave select enable feature of the SPI peripheral would do). After wiring the chip according to Fig 12.3 you can run the following sample code, which writes data to the ram and reads it back, sending the results over UART to your computer.

Code Sample 12.2. `spi_ram.c` SPI SRAM access.

```
//#define NU32_STANDALONE  // uncomment if program is standalone, not bootloaded
#include "NU32.h"          // config bits, constants, funcs for startup and UART
// Demonstrates spi by accessing external ram
// PIC is the master, ram is the slave
// Uses microchip 23K256 ram chip (see the data sheet for protocol details)
// SDO4 -> SI (pin F5 -> pin 5)
// SDI4 -> SO (pin F4 -> pin 2)
// SCK4 -> SCK (pin F13 -> pin 6)
// SS4 -> CS (pin B8 -> pin 1)
// Additional ram connections
// Vss (Pin 4) -> ground
// Vcc (Pin 5) -> 3.3v
// Hold (pin 6) -> 3.3v (we don't use the hold function)
//
// Only uses the ram's sequential mode
//
```

```
#define CS LATFbits.LATF12       // chip select pin

// send a byte via spi and return the response
unsigned char spi_io(unsigned char o) {
  SPI4BUF = o;
  while(!SPI4STATbits.SPIRBF) { // wait to receive the byte
    ;
  }
  return SPI4BUF;
}


// initialize spi4 and the ram module
void ram_init() {
  //setup the chip select pin as an output
  // the chip select pin is used by the sram to indicate
  // when a command is beginning (clear CS to low) and when it
  // is ending (set CS high)
  TRISFbits.TRISF12 = 0;
  CS = 1;

  // Setup the master Master - SPI4, pins are: SDI4(F4), SDO4(F5), SCK4(F13).
  // we manually control SS4 as a digital output (F12)
  // since the pic is just starting, we know that spi is off. We rely on defaults here

  // setup spi4
  SPI4CON = 0;              // turn off the spi module and reset it
  SPI4BUF;                  // clear the rx buffer by reading from it
  SPI4BRG = 0x3;            // baud rate to 10MHz [SPI4BRG = (80000000/(2*desired))-1]
  SPI4STATbits.SPIROV = 0;  // clear the overflow bit
  SPI4CONbits.CKE = 1;      // data changes when clock goes from active to inactive (high to low since CKP is 0)
  SPI4CONbits.MSTEN = 1;    // master operation
  SPI4CONbits.ON = 1;       // turn on spi 4

                            // send a ram set status command.
  CS = 0;                   // enable the ram
  spi_io(0x01);             // ram write status
  spi_io(0x41);             // sequential mode (mode = 0b01), hold disabled (hold = 0)
  CS = 1;                   // finish the command
}

// write len bytes to the ram, starting at the address addr
void ram_write(unsigned short addr, const char data[], int len) {
  int i = 0;
  CS = 0;                       // enable the ram
  spi_io(0x2);                  // sequential write operation
  spi_io((addr & 0xFF00) >> 8 ); //most significant byte of address
  spi_io(addr & 0x00FF);        //the least significant address byte
  for(i = 0; i < len; ++i) {
    spi_io(data[i]);
  }
  CS = 1;
}

// read len bytes from ram, starting at the address addr
void ram_read(unsigned short addr, char data[], int len) {
  int i = 0;
  CS = 0;
  spi_io(0x3);                  // ram read operation
```

```
  spi_io((addr & 0xFF00) >> 8);  // most significant address byte
  spi_io(addr & 0x00FF);         // least significant address byte
  for(i = 0; i < len; ++i) {
    data[i] = spi_io(0);         // read in the data
  }
  CS = 1;
}

int main(void) {
  unsigned short addr1 = 0x1234;                    // the test address for writing the ram
  char data[] = "Help, I'm stuck in the ram!";      // the test message
  char read[] = "******************************";  // buffer for reading from ram
  char buf[100];                                    // buffer for communication with the user
  unsigned char status;                             // used to verify that we set the status
  // setup NU32 LED's and buttons
  NU32_Startup();
  ram_init();

  // check the ram status
  CS = 0;
  spi_io(0x5);                                      // ram read status command
  status = spi_io(0);                               // the actual status
  CS = 1;

  sprintf(buf, "Status 0x%x\r\n",status);
  NU32_WriteUART1(buf);

  sprintf(buf,"Writing \"%s\" to ram at address 0x%x\r\n", data, addr1);
  NU32_WriteUART1(buf);

                                          // write the data to the ram
  ram_write(addr1, data, strlen(data) + 1);   // +1 so we write the '\0' character

  ram_read(addr1, read, strlen(data) + 1);      // read the data back
  sprintf(buf,"Read \"%s\" from ram at address 0x%x\r\n", read, addr1);
  NU32_WriteUART1(buf);

  while(1) {
    ;
  }
  return 0;
}
```

The main function used by the sample code is `spi_io`, which writes a byte to the SPI port and reads the result. Every operation uses this command to communicate with the SRAM, since every write requires a read and vice versa. After configuring the SRAM to use sequential mode, the example reads the status of the SRAM, writes some data to it, and reads it back.

After executing this sample code, comment out the write to RAM, recompile and execute the code. Notice that, if you do not power off the PIC32, the SRAM still contains the data from the previous write. Powering off the PIC32 will clear the SRAM.

12.3.3 Accelerometer/Magnetometer

The STMicroelectronics LSM303D accelerometer/magnetometer, depicted in Fig. 12.4 is a sensor that combines a 3 axis accelerometer, a 3 axis magnetometer, and a temperature sensor.[2] As a surface mount

[2]This chip uses microelectromechanical systems (MEMS) to provide you with so many sensors in such a small package.

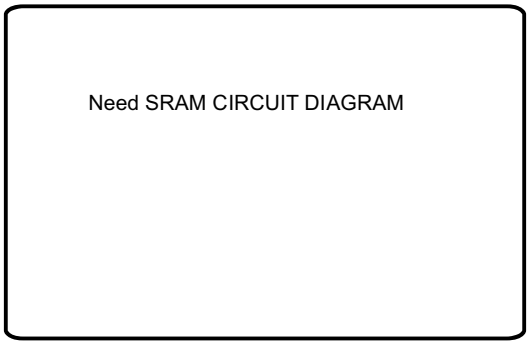


Figure 12.3: SRAM circuit diagram.

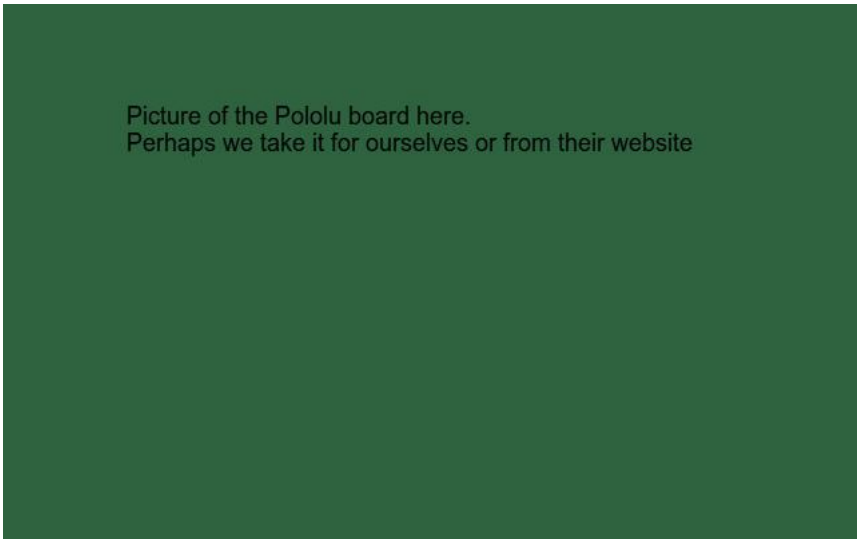


Figure 12.4: The LSM303D accelerometer on the Pololu breakout board.

component, the accelerometer is difficult to work with; therefore, we use it with a breakout board from Pololu. The combination of an accelerometer and magnetometer is ideal for creating an electronic compass: the accelerometer gives tilt parameters relative to the earth, providing a reference frame for the magnetometer readings. The PIC32 can control this device using either SPI or I^2C, another communication method discussed in Ch. **??**. You can wire the sensor according to Fig 12.5.

The example code consists of three files: `accel.h`, and `spi_accel.c`, and `accel.c`. The header file `accel.h` provides a rudimentary interface to the accelerometer; it can be implemented either using SPI or I^2C. The actual SPI code required is provided by `spi_accel.c`. The SPI peripheral is set for 10 Mhz operation and, as in `spi_ram.c`, `spi_io` encapsulates the write/read behavior. The `main` function is implemented in `accel.c`, which reads and displays the sensor values approximately once per second.

Code Sample 12.3. `accel.h` High-level interface to the LSM303D accelerometer/magnetometer

```
#ifndef ACCEL__H__
#define ACCEL__H__
// Basic interface with an LSM303D accelerometer/compass.
// Used for both i2c and spi examples, but with different implementation (.c) files

                        // register addresses
#define CTRL1 0x20      // control register 1
#define CTRL5 0x24      // control register 5
```

```
#define CTRL7 0x26      // control register 7

#define OUT_X_L_A 0x28  // LSB of x axis acceleration register.
                        // all acceleration registers are contiguous, and this is the lowest address
#define OUT_X_L_M 0x08  // LSB of x axis of magnetometer register

#define TEMP_OUT_L 0x05 // temperature sensor register

// read len bytes from the specified register into data[]
void acc_read_register(unsigned char reg, unsigned char data[], unsigned int len);

// write to the register
void acc_write_register(unsigned char reg, unsigned char data);

// initialize the accelerometer
void acc_setup();
#endif
```

Code Sample 12.4. `accel.c` Accelerometer/magnetometer test code.

```
#include "accel.h"
#include "NU32.h"
// accelerometer/magnetometer example.  Prints the results from the sensor to the UART

int main() {
  char buffer[200];
  NU32_Startup();
  acc_setup();

  short accels[3];   // accelerations for the 3 axes
  short mags[3];     // magnetometer readings for the 3 axes
  short temp;
  while(1) {
    // read the accelerometer from all three axes
    // the accelerometer and the pic32 are both little endian by default (the lowest address has the LSB)
    // the accelerations are 16-bit twos compliment numbers, the same as a short
    acc_read_register(OUT_X_L_A, (unsigned char *)accels,6);

    // NOTE: the accelerometer reads DC accelerations
    // (meaning that, on earth, when stationary it measures gravity as a 1 g acceleration)
    // You could use this information to calibrate the readings into actual units
    sprintf(buffer,"x: %d y: %d z: %d\r\n",accels[0], accels[1], accels[2]);
    NU32_WriteUART1(buffer);

    // need to read all 6 bytes in one transaction to get an update. e
    acc_read_register(OUT_X_L_M, (unsigned char *)mags, 6);

    sprintf(buffer, "xmag: %d ymag: %d zmag: %d \r\n",mags[0], mags[1], mags[2]);
    NU32_WriteUART1(buffer);

    // read the temperature data.  Its a right justified 12 bit two's compliment number
    acc_read_register(TEMP_OUT_L,(unsigned char *)&temp,2);
    sprintf(buffer,"temp: %d\r\n",temp);
    NU32_WriteUART1(buffer);

    //delay
   _CP0_SET_COUNT(0);
```

```
   while(_CP0_GET_COUNT() < 40000000) { ; }
  }
}
```

Code Sample 12.5. spi_accel.c Communicates with the LSM303D accelerometer/magnetometer using SPI.

```
#include "accel.h"
#include "NU32.h"
// interface with the LSM303D accelerometer/magnetometer using spi
// Wire GND to GND, VDD to 3.3V,
// SDO4 (F5) ->  SDI (labeled SDA),
// SDI4 (F4) -> SDO
// SCK4 (F13) -> SCL
// RF12 -> CS
#define CS LATFbits.LATF12        // use RF12 as CS

// send a byte via spi and return the response
unsigned char spi_io(unsigned char o) {
  SPI4BUF = o;
  while(!SPI4STATbits.SPIRBF) { // wait to receive the byte
    ;
  }
  return SPI4BUF;
}


// read data from the accelerometer, given the starting register address.
// return the data in data
void acc_read_register(unsigned char reg, unsigned char data[], unsigned int len)
{
  unsigned int i;
  reg |= 0x80; // set the read bit (as per the accelerometer's protocol)
  if(len > 1) {
    reg |= 0x40; // set the address auto increment bit (as per the accelerometer's protocol)
  }
  CS = 0;
  spi_io(reg);
  for(i = 0; i != len; ++i) {
    data[i] = spi_io(0); // read data from spi
  }
  CS = 1;
}


void acc_write_register(unsigned char reg, unsigned char data)
{
  CS = 0;               // bring CS low to activate SPI
  spi_io(reg);
  spi_io(data);
  CS = 1;               // complete the command
}

void acc_setup() {              // setup the accelerometer, using spi 4
  TRISFbits.TRISF12 = 0;
  CS = 1;
```

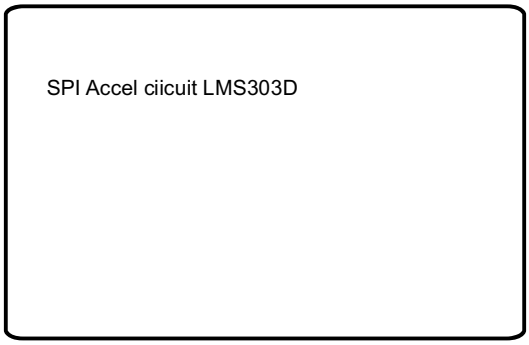


Figure 12.5: Circuit diagram for the LSM303D, when used with SPI.

```
  // Setup the master Master - SPI4, pins are: SDI4(F4), SDO4(F5), SCK4(F13).
  // we manually control SS4 as a digital output (F12)
  // since the pic is just starting, we know that spi is off. We rely on defaults here

  // setup spi4
  SPI4CON = 0;              // turn off the spi module and reset it
  SPI4BUF;                  // clear the rx buffer by reading from it
  SPI4BRG = 0x3;            // baud rate to 10MHz [SPI4BRG = (80000000/(2*desired))-1]
  SPI4STATbits.SPIROV = 0;  // clear the overflow bit
  SPI4CONbits.CKE = 1;      // data changes when clock goes from active to inactive
                            //    (high to low since CKP is 0)
  SPI4CONbits.MSTEN = 1;    // master operation
  SPI4CONbits.ON = 1;       // turn on spi 4

  // set the accelerometer data rate to 1600 Hz. Do not update until we read values
  acc_write_register(CTRL1, 0xAF);

  // 50 Hz magnetometer, high resolution, temperature sensor on
  acc_write_register(CTRL5, 0xF0);

  // enable continuous reading of the magnetometer
  acc_write_register(CTRL7, 0x0);
}
```

You can test the accelerometer readings by tilting the board in various directions. The accelerometer reads DC accelerations; therefore it will always read 1g of acceleration in the downward direction. If you rotate the board you should see the magnetic field readings change. You can test the temperature sensor by blowing on it: the heat from your breath will cause the reading to temporarily increase. To use this device as a magnetic compass you must perform a calibration: STMicroelectronics application note AN3192 provides a guide.

12.4 Chapter Summary

- SPI devices can be either a master or a slave. Usually, the PIC32 is configured as the master.
- Two-way communication usually requires four wires: clock (SCK), input (SDI), output (SDO), and a chip select (SS). rather than having the SPI peripheral manage it for you.
- Writes and reads happen simultaneously. Every write should be followed by a read, even if you do not need the data.

- The master generates the clock and therefore determines the frequency of communication. The slave can only send data when it receives a clock signal. The master generates a clock signal when you write to SPIxBUF. To read data, you must first write to SPIxBUF to generate a clock signal, even if the slave device does not need any data.

12.5 Exercises

Chapter 13

I^{2}C Communication

Inter-integrated circuit (I^{2}C) allows the PIC32 to communicate with multiple devices using only two pins. Many devices have I^{2}C interfaces, including RAM, accelerometers, ADCs, and OLED screens. Some advantages of I^{2}C are that it requires only two wires and that multiple master and slave devices can be connected to the same bus, further reducing the number of pins required. Disadvantages include the relatively slow bit-rates, complicated software processing, and need for external pull-up resistors.

13.1 Overview

The I^{2}C bus consists of two wires, one for data (SDA) and one for a clock (SCL) (chips must also have a common ground reference). Multiple chips can be connected to the same two wires and communicate with each other. A chip can be a master or a slave. Multiple masters and slaves can be connected to the same bus; however, only one master can operate at one time. Masters control the clock signal and hence the speed at which data flows. Usually, I^{2}C devices operate either in standard 100 kHz or high speed 400 kHz mode, although the PIC32 can set arbitrary bit-rates.

Figure 13.1 shows a circuit diagram of a typical bus connection. Rather than driving the lines high and low, each pin on the I^{2}C bus switches between low and a high impedance (disconnected, floating, high-z) state. The high-z state is equivalent to the open-drain digital output mode, where the pin, rather than being high, is effectively disconnected. Pull-up resistors on each I^{2}C line ensure that the line is low only when a device outputs a low signal. Using this mechanism, if a device outputs a high signal but detects a low signal, it knows another device is using the bus. This mechanism allows masters in multi-master systems to detect when other masters are attempting to use the bus, a condition known as a bus collision. It also allows slave devices to pause a transfer to allow more processing time by holding the clock line low, a process called clock stretching: masters detect this condition and pause the clock signal until SCL is high.

Unlike UART or SPI, I^{2}C uses a specific low-level bus protocol to communicate, depicted in 13.2. A master first sends a start bit, signaling the beginning of a transmission. It then sends an address byte. Each slave device has an address and only responds when it detects its own address. In standard mode the address byte consists of a 7-bit slave address and a read/write bit. There is also a 10-bit address mode, but we assume 7-bit addresses throughout this chapter. The read/write bit determines if the next data will be written to or read from the slave. To change the read/write direction on the bus, the master can issue a restart bit, followed by the slave address and read/write bit.

13.1.1 The Protocol

We now provide you with an overview of the I^{2}C protocol, as depicted in 13.2. We only cover simple read and write transactions. More advanced details can be found in the I^{2}C section of the reference manual and the I^{2}C standard [**?**]. You will also need to consult the data sheet of the slave device to know what data to send and receive. For the purposes of this discussion, we omit details regarding the timing and electrical traits of the signals, favoring a software oriented approach.

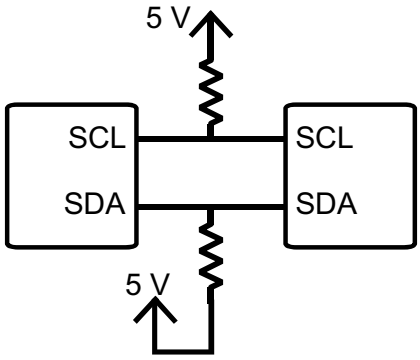
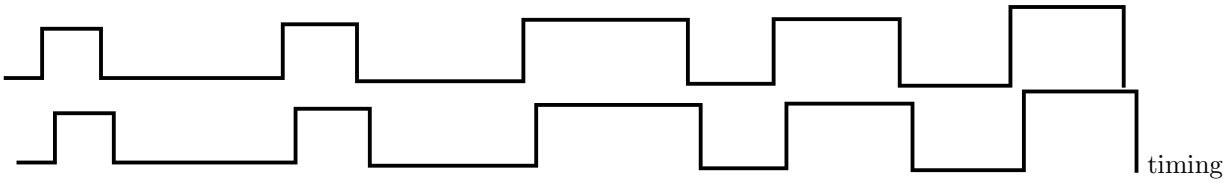


Figure 13.1: The I²C bus



timing for SCL and SDA lines. start bit. then 7 bit address with R/W low, then data, the restart then address w/ R/W high. Also include acks Should show transition between between when slave and when master is modifying SDA

Figure 13.2: A write-read sequence on the I²C bus.

The I²C protocol consists of a series of primitive commands that can be combined to form a complete I²C transaction. The I²C peripheral handles the signaling details for each primitive and software coordinates the overall transaction by initiating primitives and waiting for them to complete. The primitive operations are described below.

- START: Begins a transaction on the I²C bus.
- ADDRESS: Chooses the slave device and read/write direction. Typically it is one byte long (7-bit address and 1 read/write bit), but 10-bit addresses are also supported.
- WRITE: Master writes a byte of data to the slave.
- RECEIVE: The master reads a byte of data from the slave.
- ACK: Sent from the master to indicate that a slave should send more data. Sent from a slave to indicate receipt of data.
- NACK: Sent from the master to indicate that a slave should not send data. The master can receive a NACK if the slave does not respond, indicating an error.
- RESTART: Sent by the master, allows changing the read/write direction. After issuing a RESTART an ADDRESS should be issued.
- STOP: Terminates the I²C transaction.

A transaction begins with the master issuing a START, followed by an ADDRESS. The transaction ends with a STOP. Between the ADDRESS and STOP, other primitives can be issued, depending on the specific slave device. Typical transactions are writes, reads, and write-read combinations. A write consists of the following steps

1. Master sends a START to begin the transaction.

2. Master sends an ADDRESS byte. The byte consists of an address in bits 7-1 and a zero in the lsb (bit 0), indicating a write.
3. Master verifies that the slave sent an ACK. If the check fails, an error has occurred.
4. Master issues a WRITE, sending a data byte. It continues sending bytes until it is done.
5. After each WRITE, the master checks for an ACK from the slave. Failing to receive an ACK indicates an error.
6. Master sends a STOP, to end the transaction.

To read from the slave device, the master does the following:

1. Master sends a START, to begin the transaction.
2. Master sends an ADDRESS byte. The byte consists of and address in bits 7-1 and a one in the LSB (bit 0), indicating a read.
3. Master checks that a slave sent an ACK. If the check fails, an error has occurred.
4. Master initiates a RECEIVE command, providing the slave with a clock signal. The slave sends a byte in response to the clock.
5. Master sends an ACK if it wants another byte or a NACK if it is done requesting bytes.
6. When finished receiving the bytes, master sends a STOP.

A write-read command starts as a normal write command. Instead of issuing a STOP, the master sends a RESTART. The RESTART replaces the START expected in a normal read command, and the transaction continues with the ADDRESS, just like a normal read. Essentially the RESTART allows the master to change the direction of the transaction without relinquishing bus control, an issue in multi-master setups.

13.2 Details

Seven SFRs control the behavior of I²C peripherals. Many of the fields in these SFRs initiate an "event" on the bus.

I2CxCON The I²C control register. Setting bits in this register initiates primitive operations used by the I²C protocol. Some bits control an I²C slave's behavior.

I2CxCON⟨15⟩ or I2CxCONbits.ON: Setting this bit enables the I²C module.

I2CxCON⟨12⟩ or I2CxCONbits.SCLREL: Set by the slave to signal that it is done processing and communication may continue. Hardware automatically clears the bit before the slave is expected to transmit and, optionally (if I2CxCONbits.STREN = 1) after the slave receives data. When zero, the slave holds the SCL line low, a process called clock stretching. When the master detects that SCL is low it delays sending a clock signal, giving the slave more time to perform computations prior to responding to the master.

I2CxCON⟨6⟩ or I2CxCONbits.STREN: Used by the slave to control clock stretching. If set to one, the slave will hold SCL low prior to transmitting to and after receiving from the master. When SCL is low, the clock is stretched and the master pauses the clock. If clear, clock stretching only happens at the beginning of a slave transmission. A slave transmission begins whenever the master expects data from the slave, according to the I²C protocol. The transmission does not actually occur until the slave sets I2CxCONbits.SCLREL.

I2CxCON⟨5⟩ or I2CxCONbits.ACKDT: Acknowledge data bit. If set to one, the master will send a NACK when sending an acknowledgment, which signals to the slave to send more data. If set to zero, the master sends an $\overline{\text{NACK}}$ during the acknowledge, signaling that it wants more data.

I2CxCON⟨4:0⟩ These bits initiate various control signals on the bus. While any of these bits is high, you should not set any of the other bits.

- I2CxCON⟨4⟩ or I2CxCONbits.ACKEN: Setting this bit initiates an ACK. Hardware clears this bit after the ACK ends.
- I2CxCON⟨3⟩ or I2CxCONbits.RCEN: Setting this bit initiates a RECEIVE. Hardware clears this bit after the receive has finished.
- I2CxCON⟨2⟩ or I2CxCONbits.PEN: Setting this bit initiates a STOP. Hardware clears the bit after the stop is sent.
- I2CxCON⟨1⟩ or I2CxCONbits.RSEN: Setting this bit initiates a RESTART. Hardware clears this bit after the restart is finished.
- I2CxCON⟨0⟩ or I2CxCONbits.SEN: Setting this bit initiates a START. Hardware clears this bit after the start is finished.

I2CxSTAT Contains the status of of the I²C peripheral and results from signals on the I²C bus.

- I2CxSTAT⟨15⟩ or I2CxSTATbits.ACKSTAT: If clear (0) an ACK (acknowledge) has been received; otherwise an ACK has not been received.
- I2CxSTAT⟨14⟩ or I2CxSTATbits.TRSTAT: If set (1) the master is transmitting; otherwise the master is not transmitting. Only used in master mode.
- I2CxSTAT⟨6⟩ or I2CxSTATbits.I2COV: Useful for debugging. If set (1), a receive overflow has occurred, meaning the receive buffer contains a byte but another byte has been received. Software must clear this bit.
- I2CxSTAT⟨5⟩ or I2CxSTATbits.DA: Used by the slave to determine if the byte is data (1) or an address (0).
- I2CxSTAT⟨2⟩ or I2CxSTATbits.RW: Used by the slave to determine if data transfer is a read (output from the slave, 1) or a write (input to the slave, 0).
- I2CxSTAT⟨1⟩ or I2CxSTATbits.RBF: The receive buffer full bit, when set, indicates that a byte has been received and is ready in I2CxRCV.
- I2CxSTAT⟨0⟩ or I2CxSTATbits.TBF: When set (1), indicates that the transmit buffer is full and that a transmission is occurring.

I2CxADD This register contains the slave's address. The address is contained in the first 10 bits (although only 7 are used in 7-bit address mode). By setting a slave address the I²C peripheral acts as a slave. Whenever an ADDRESS is initiated on the I²C bus, the slave will respond if the ADDRESS matches its own address.

I2CxMSK Allows the slave to ignore some address bits.

I2CxBRG The lower 16 bits of this register determine the baud. Typically the baud is either 100 kHz or 400 kHz. To compute the value of I2CxBRG, use the formula

$$\text{I2CxBRG} = [(\frac{1}{2 * F_{sck}} - T_{PGD})PBCLK] - 2, \tag{13.1}$$

where F_{sck} is the desired baud, PBCLK is the peripheral bus clock period, and T_PGD is 104 ns. With an 80 MHz peripheral bus clock, I2CxBRG = 390 for 100 kHz and I2CxBRG = 90 for 400 kHz.

I2CxTRN Used to transmit a byte of data. Write an address to this register when performing the ADDRESS command, or write data when sending a data byte.

I2CxRCV Used to receive data from the I²C bus. This register contains any data received. On the master, this register only contains data after a RECEIVE request has been initiated. On the slave, this register is loaded whenever the master sends any data, including address bytes.

13.3 Sample Code

13.3.1 Loopback

In this example, a single PIC32 communicates with itself using I²C. Here we use I²C 2 as the master and I²C five as a slave.[1]

We have divided the code into three modules: the master, the slave, and the main function, allowing you to test the code on a single PIC and then easily divide the master and slave to test inter-PIC communication.

First, the master code.

Code Sample 13.1. `i2c_master_noint.h` Header file for I²C master with no interrupts.

```
#ifndef I2C_MASTER_NOINT_H__
#define I2C_MASTER_NOINT_H__
// Header file for i2c_master_noint.c
// helps implement use i2c2 as amaster without using interrupts

void i2c_master_setup(void);    // setup i2c 2 as a master, at 100 kHz

void i2c_master_start(void);    // send a START signal
void i2c_master_restart(void);  // send a RESTART signal
void i2c_master_send(unsigned char byte); // send a byte (either an address or data)
unsigned char i2c_master_recv(void);      // receive a byte of data
void i2c_master_ack(int val);             // send an ACK (0) or NACK (1)
void i2c_master_stop(void);               // send a stop

#endif
```

Code Sample 13.2. `i2c_master_noint.c` Implementation of I²C master with no interrupts.

```
#include"NU32.h"
// I2C Master utilities, using polling rather than interrupts
// Master will use I2C2 SDA2 (A3) and SCL2 (A2)
// Connect these through resistors to Vcc (3.3V). 2.3k resistors recommended, but something close will do.
// Connect SDA2 to the SDA pin on a slave device and SCL2 to the SCL pin on a slave device
// Some utilities for operating an i2c master bus in polled mode.
// The functions must be callled in the correct order as per the i2c protocol

void i2c_master_setup(void) {
  I2C2BRG = 390;                    // I2CBRG = [1/(2*Fsck) - PGD]*Pblck - 2
                                    // Fsck is the frequency (usually 100khz or 400 khz), PGD = 104ns
  I2C2CONbits.ON = 1;               // turn on the I2C2 module
}

// Start a transmission on the i2c bus
void i2c_master_start(void) {
    I2C2CONbits.SEN = 1;        // send the start bit
    while(I2C2CONbits.SEN) {
      ;                         // wait for the start bit to be sent
    }
}

                                  // Send a restart.
                                  // Used for changing I/O direction without relinquishing bus control
```

[1] Technically, an I²C peripheral can simultaneously be a master and a slave. When the master sends data to its slave address, the slave will respond!

```
void i2c_master_restart(void) {
    I2C2CONbits.RSEN = 1;           // send a restart enable bit. (Restart allows switching
                                    // between reading and writing without giving up control
                                    // of the I2C bus (relevant when multiple masters are attached).
    while(I2C2CONbits.RSEN) {       // wait for the restart to clear
      ;
    }
}

                                    // Send a byte to the slave
                                    // If this is an address byte, then bit 0 is 1 for Read, 0 for Write
void i2c_master_send(unsigned char byte) {
  I2C2TRN = byte;
  while(I2C2STATbits.TRSTAT) {
   ;                                //  wait for the transmission to finish
  }
  if(I2C2STATbits.ACKSTAT) {        // the slave acknowledges by sending a low bit, so if this is high
                                    // the slave has not acknowledged the transmission
    NU32_WriteUART1("I2C2 Master: failed to receive ACK\r\n");
  }
}

unsigned char i2c_master_recv(void) {     // receive a byte from the slave
    I2C2CONbits.RCEN = 1;             // start receiving data

    while(!I2C2STATbits.RBF) {        // wait to receive the data
      ;
    }
    return I2C2RCV;                   // read and return the data
}

                                    // sends an ack (val == 0) or nack (val == 1) to the slave.
                                    // an ack tells the slave to send another byte and nack means
                                    // do not send another byte
void i2c_master_ack(int val) {
    I2C2CONbits.ACKDT = val;          // send an ACK/NACK, to tell the slave to send/not send another byte
    I2C2CONbits.ACKEN = 1;
    while(I2C2CONbits.ACKEN) {
      ;                               // wait for the NACK to be sent
    }
}

                                    // send a stop signal, indicating that the
                                    // communication is complete and the master is
                                    // relinquishing bus control
void i2c_master_stop(void) {
  I2C2CONbits.PEN = 1;              // send the stop
  while(I2C2CONbits.PEN) {
    ;                               // wait to be done sending the stop
  }
}
```

The implementation of the I²C master contains functions roughly corresponding to the primitives that the hardware can perform. Each function executes the primitive command and waits for it to complete. By calling the primitive functions in succession, you can form an I²C transaction.

Next the slave code.

Code Sample 13.3. `i2c_slave.h` Header file for I²C slave.

```
#ifndef I2C_SLAVE_H__
#define I2C_SLAVE_H__
// implements a basic i2c slave

void i2c_slave_setup(unsigned char addr); // setup the slave at the given address

#endif
```

Code Sample 13.4. `i2c_slave.c` Implementation of an I²C slave.

```
#include"NU32.h"
// Implements a I2C slave on I2C5 using pins SDA5 (F4) and SCL5 (F5)
// The slave returns the last two bytes the master writes

void __ISR(_I2C_5_VECTOR, IPL1SOFT) I2C5SlaveInterrupt(void) {
  static unsigned char bytes[2];    // store the bytes we receive
  static int rw = 0;                // index of the bytes read/written
  if(rw == 2) {
    rw = 0;
  }
                                    // determine why the slave interrupt occurred
  if(I2C5STATbits.D_A) {            // the byte is a data byte
    if(I2C5STATbits.R_W) {          // the master expects data
      I2C5TRN = bytes[rw];          // send the master expected data
      I2C5CONbits.SCLREL = 1;       // release the clock, allowing the master to clock in data
    } else {                        // the master has sent data
      bytes[rw] = I2C5RCV;          // read the data the master has sent
    }
    ++rw;
  } else {                          // the byte is an address byte
    I2C5RCV;                        // read the address to clear the buffer
    if(I2C5STATbits.R_W) {          // the master expects data
      I2C5TRN = bytes[rw];
      ++rw;
      I2C5CONbits.SCLREL = 1;       // release the clock, allowing the master to clock in data
    }
                                    // otherwise, an address has been written so do nothing
  }
  IFS1bits.I2C5SIF = 0;
}

void i2c_slave_setup(unsigned char addr) {       // setup I2C5 as a slave (disable interrupts before calling this)
  I2C5ADD = addr;                  // the address of the slave

  IPC8bits.I2C5IP  = 1;            // slave has interrupt priority 1
  IEC1bits.I2C5SIE = 1;            // slave interrupt is enabled
  IFS1bits.I2C5SIF = 0;            // clear the interrupt flag
  I2C5CONbits.ON   = 1;            // turn on i2c2
}
```

The slave code uses I²C 5, and, upon a read request, will return the last two bytes written to the slave. As the slave is interrupt driven, it will work as soon as you call `i2c_slave_setup`, providing the desired 7-bit address (the master is configured to talk to a slave on address 0x32). The slave interrupt reads the status flag so that it can discriminate between reads, writes, address bytes, and data bytes. Notice that, whenever the slave sends data to the master, it must release the clock by setting I2C5CONbits.SCLREL to one.

Next, the main program, in which the PIC32 communicates with itself over I²C.

Code Sample 13.5. `i2c_loop.c` The main I²C loopback program.

```
#include "NU32.h"
#include "i2c_slave.h"
#include "i2c_master_noint.h"
// Demonstrate I2C by having the I2C2 talk to I2C5 on the same PIC
// Master will use SDA2 (A3) and SCL2 (A2).  Connect these through
// resistors to Vcc (3.3V) (recommended is a 2.3k resistor, but around that should be good enough)
// Slave will use SDA5 (F4) and SCL5 (F5)
// SDA5 -> SDA2
// SCL5 -> SCL2
// Two bytes will be written to the slave and then read back to the slave.
#define SLAVE_ADDR 0x32

int main() {
  char buf[100] = {};                       // buffer for sending messages to the user
  unsigned char master_write0 = 0xCD;       // first byte to write
  unsigned char master_write1 = 0x91;       // second byte to write
  unsigned char master_read0  = 0x00;       // first received byte
  unsigned char master_read1  = 0x00;       // second received byte

  NU32_Startup();
  __builtin_disable_interrupts();
  i2c_slave_setup(SLAVE_ADDR);              // init I2C5, which we use as a slave
                                            //  (comment out if slave is on another pic)
  i2c_master_setup();                       // init I2C2, which we use as a master
  __builtin_enable_interrupts();

  while(1) {

    NU32_WriteUART1("Master: Press Enter to begin transmission.\r\n");
    NU32_ReadUART1(buf,2);

    i2c_master_start();                     // Begin the start sequence

    i2c_master_send(SLAVE_ADDR << 1);       // send the slave address, left shifted by 1,
                                            // which clears bit 0, indicating a write
                                            //
    i2c_master_send(master_write0);         // send a byte to the slave

    i2c_master_send(master_write1);         // send another byte to the slave

    i2c_master_restart();                   // send a restart so we can begin reading

    i2c_master_send((SLAVE_ADDR << 1) | 1); // send a read command (the 0 bit is 1 indicating read)

    master_read0 = i2c_master_recv();       // receive a byte from the bus

    i2c_master_ack(0);                      // send an ack (0), indicating that the master wants another byte

    master_read1 = i2c_master_recv();       // receive another byte from the bus

    i2c_master_ack(1);                      // send a nack (1), indicating that there are no more reads

    i2c_master_stop();                      // send a stop, ending transmission and relinquishing bus control
```

```
    sprintf(buf,"Master Wrote: 0x%x 0x%x\r\n", master_write0, master_write1);
    NU32_WriteUART1(buf);

    sprintf(buf,"Master Read: 0x%x 0x%x\r\n", master_read0, master_read1);
    NU32_WriteUART1(buf);
    ++master_write0;
    ++master_write1;
  }

  return 0;
}
```

The main program initializes the slave and master and performs some I²C transactions, sending the results over the UART to your terminal. Notice how the primitive operations are assembled into a single transaction. You should connect the clock (SCL) and data (SDA) lines of I²C 2 and I²C 5, along with the pull-up resistors. To examine what happens when the slave does not return an ACK, disconnect the I²C bus wires.

Using I²C to have the PIC32 communicate with itself may not seem practical, but it helps demonstrate the peripheral without involving other chips. If you have another PIC32 available, you can compile the slave as a standalone program. By connecting the master loopback example to a slave on another chip you can witness inter-PIC communication.

Code Sample 13.6. `i2c_slave_loop.c` A standalone I²Cslave.

```
#include "i2c_slave.h"
#include "NU32.h"

int main() {
  NU32_Startup();
  i2c_slave_setup(0x32); // enable the slave w/ address 0x32

  while(1) {  // the slave is handled in an interrupt in i2c_slave.c
    _nop();   // so we do nothing.
  }
  return 0;
}
```

To use `i2c_slave_loop.c`, compile and link it with `i2c_slave.c` and run it on another PIC32. Connect the SDA and SCL wires between the two PIC32s. Power the second PIC32 from the first PIC32 by connecting the GND and 6 volt Vin supplies together. Plug-in the master PIC32 while making sure that the slave PIC32 is unplugged. Both PIC32's will turn on and off based on the state of the master's switch.

13.3.2 Interrupt-based Master

In Sec 13.3.1 we created functions for each I²C primitive: the functions initiate a command and wait for it to complete. In this section, we provide interrupt-based master code. The function `i2c_write_read` allows the user to initiate a write-read transaction by providing a slave address, an input array with the bytes to write, and an output array with the bytes that are read. If the length of the write or read array is zero, then that specific action will not be performed.

Code Sample 13.7. `i2c_master_int.h` Header file for interrupt-based I²C master.

```
#ifndef I2C_MASTER_INT__H__
#define I2C_MASTER_INT__H__

// buffer pointer type.  The buffer is shared by an ISR and mainline code.
// the pointer to the buffer is also shared by an ISR and mainline code.
```

```
// Hence the double volatile qualification
typedef volatile unsigned char * volatile buffer_t;


void i2c_master_setup(); //sets up i2c2 as a master using an interrupt

// initiate an i2c write read operation at the given address.
// You can optionally only read or only write by passing zero lengths for the reading or writing
// this will not return until the transaction is complete.  returns false on error
int i2c_write_read(unsigned int addr, const buffer_t write, unsigned int wlen,
    const buffer_t read, unsigned int rlen );

#endif
```

Code Sample 13.8. `i2c_master_int.c` Implementation of an interrupt-based I²C master.

```
#include "i2c_master_int.h"
#include "NU32.h"
// I2C Master utilities, using interrupts
// Master will use I2C2 SDA2 (A3) and SCL2 (A2)
// Connect these through resistors to Vcc (3.3V). 2.3k resistors recommended, but something close will do.
// Connect SDA2 to the SDA pin on a slave device and SCL2 to the SCL pin on a slave device

// keeps track of the current i2c state
static volatile enum {IDLE, START, WRITE, READ,RESTART,ACK,NACK, STOP, ERROR} state = IDLE;

static buffer_t to_write = NULL;  // data to write
static buffer_t  to_read = NULL;  // data to read
static volatile unsigned char address = 0;         // the 7-bit address to write to / read from
static volatile unsigned int n_write = 0;          // number of data bytes to write
static volatile unsigned int n_read = 0;           // number of data bytes to read


void __ISR(_I2C_2_VECTOR, IPL1SOFT) I2C2SlaveInterrupt(void) {
  static unsigned int write_index = 0, read_index = 0;   // indexes into the read/write arrays

  switch(state) {
    case START:                                 // start bit has been sent
      write_index = 0;                          // reset indices
      read_index = 0;
      if(n_write > 0) {                         // there are bytes to write
        state = WRITE;                          // transition to write mode
        I2C2TRN = address << 1;                 // send the address, with write mode set
      } else {
        state = ACK;                            // skip directly to reading
        I2C2TRN = (address << 1) & 1;
      }

      break;
    case WRITE:                                  // a write has finished
      if(I2C2STATbits.ACKSTAT) {                 // did not receive a nack from the slave, this is an error
        state = ERROR;
      } else {
        if(write_index < n_write) {              // still more data to write
          I2C2TRN = to_write[write_index];       // write the data
          ++write_index;
        } else {                                 // done writing data, time to read or stop
```

```
          if(n_read > 0) {                    // we want to read so issue a restart
            state = RESTART;
            I2C2CONbits.RSEN = 1;             // send the restart to begin the read
          }
          else {                              // no data to read, issue a stop
            state = STOP;
            I2C2CONbits.PEN = 1;
          }
        }
      }
      break;
    case RESTART: // the restart has completed
      // now we want to read, send the read address
      state = ACK;                  // when interrupted in ACK mode, we will initiate reading a byte
      I2C2TRN = (address << 1) | 1; // the address is sent with the read bit sent
      break;
    case READ:
      to_read[read_index] = I2C2RCV;
      ++read_index;
      if(read_index == n_read) { // we are done reading, so send a nack
        state = NACK;
        I2C2CONbits.ACKDT = 1;
      } else {
        state = ACK;
        I2C2CONbits.ACKDT = 0;
      }
      I2C2CONbits.ACKEN = 1;
      break;
    case ACK:
              // just sent an ack meaning we want to read more bytes
      state = READ;
      I2C2CONbits.RCEN = 1;
      break;
    case NACK:
      //issue a stop
      state = STOP;
      I2C2CONbits.PEN = 1;
      break;
    case STOP:
      state = IDLE; // we have returned to idle mode, indicating that the data is ready
      break;
    default:
      // some error has occurred
      state = ERROR;
  }
  IFS1bits.I2C2MIF = 0;       //clear the interrupt flag
}

void i2c_master_setup() {
  int ie = __builtin_disable_interrupts();
  I2C2BRG = 90;                    // I2CBRG = [1/(2*Fsck) - PGD]*Pblck - 2
                                   // Fsck is the frequency (usually 100khz or 400 khz), PGD = 104ns
                                   // this is 400 khz mode
                                   // enable the i2c interrupts
  IPC8bits.I2C2IP  = 1;            // master has interrupt priority 1
  IEC1bits.I2C2MIE = 1;            // master interrupt is enabled
  IFS1bits.I2C2MIF = 0;            // clear the interrupt flag
  I2C2CONbits.ON = 1;              // turn on the I2C2 module
```

```
  if(ie & 1) {
    __builtin_enable_interrupts();
  }
}

// communicate with the slave at address addr.  first write wlen bytes to the slave, then read
// rlen bytes from the slave
int i2c_write_read(unsigned int addr, const buffer_t write,
    unsigned int wlen, const buffer_t read, unsigned int rlen ) {
  n_write = wlen;
  n_read = rlen;
  to_write = write;
  to_read = read;
  address = addr;
  state = START;
  I2C2CONbits.SEN = 1;        // initialize the start
  while(state != IDLE && state != ERROR) { ; }  // initialize the sequence
  return state != ERROR;
}

// write a single byte to the slave
int i2c_write_byte(unsigned int addr, unsigned char byte) {
  return i2c_write_read(addr,&byte,1,NULL,0);
}
```

The interrupt removes the need to wait for each primitive operation to complete: rather the interrupt triggers at the end of each primitive. The ISR tracks the state of the current communication and performs the appropriate state transition. As coded, the function `i2c_write_read` waits for the whole transaction to complete before returning; however, in time critical applications this behavior could be modified. Calling `i2c_write_read` would initiate the transaction, but not wait for it to finish. Mainline code could continue to execute, and either check for the result of the transaction at a later time or handle the transaction results from within the ISR.

13.3.3 Accelerometer/Magnetometer

The STMicroelectronics LSM303D is a 3 axis accelerometer and magnetometer that can be used as a digital compass. It also contains a temperature sensor. More details about this sensor and the breakout board are discussed in Ch. 12. Here we use the same accelerometer library and example that we used in Ch. 12 except now our implementation uses I²C rather than SPI. For convenience we list all of the necessary code here.

Code Sample 13.9. `accel.h` Basic interface to the LSM303D accelerometer/magnetometer.

```
#ifndef ACCEL__H__
#define ACCEL__H__
// Basic interface with an LSM303D accelerometer/compass.
// Used for both i2c and spi examples, but with different implementation (.c) files

                        // register addresses
#define CTRL1 0x20      // control register 1
#define CTRL5 0x24      // control register 5
#define CTRL7 0x26      // control register 7

#define OUT_X_L_A 0x28  // LSB of x axis acceleration register.
                        // all acceleration registers are contiguous, and this is the lowest address
#define OUT_X_L_M 0x08  // LSB of x axis of magnetometer register
```

```
#define TEMP_OUT_L 0x05 // temperature sensor register

// read len bytes from the specified register into data[]
void acc_read_register(unsigned char reg, unsigned char data[], unsigned int len);

// write to the register
void acc_write_register(unsigned char reg, unsigned char data);

// initialize the accelerometer
void acc_setup();
#endif
```

Code Sample 13.10. `i2c_accel.c` I²C implementation of the basic accelerometer library. Requires `i2c_master_int.h`

```
#include "accel.h"
#include "i2c_master_int.h"
#include <stdlib.h>

#define I2C_ADDR 0x1D // the I2C slave address

// Wire GND to GND, VDD to 3.3V, SDA to SDA2 (RA3) and SCL to SCL2 (RA2)

// read data from the accelerometer, given the starting register address.
// return the data in data
void acc_read_register(unsigned char reg, unsigned char data[], unsigned int len)
{
  unsigned char write_cmd[1] = {};
  if(len > 1) { // want to read more than 1 byte and we are reading from the accelerometer
    write_cmd[0] = reg | 0x80; // make the MSB of addr 1 to enable auto increment
  }
  else {
    write_cmd[0] = reg;
  }
  i2c_write_read(I2C_ADDR,write_cmd, 1, data,len);
}


void acc_write_register(unsigned char reg, unsigned char data)
{
  unsigned char write_cmd[2];
  write_cmd[0] = reg;   // write the register
  write_cmd[1] = data;  // write the actual data
  i2c_write_read(I2C_ADDR, write_cmd, 2, NULL, 0);
}

void acc_setup() {              // setup the accelerometer, using I2C 2
  i2c_master_setup();
  acc_write_register(CTRL1, 0xAF); // set accelerometer data rate to 1600 Hz. Don't update until we read values
  acc_write_register(CTRL5, 0xF0); // 50 Hz magnetometer, high resolution, temperature sensor on
  acc_write_register(CTRL7, 0x0); // enable continuous reading of the magnetometer
}
```

Code Sample 13.11. `accel.c` Example code that reads the LSM303D and prints the results over UART.

```
#include "accel.h"
#include "NU32.h"
// accelerometer/magnetometer example.  Prints the results from the sensor to the UART

int main() {
  char buffer[200];
  NU32_Startup();
  acc_setup();

  short accels[3];   // accelerations for the 3 axes
  short mags[3];     // magnetometer readings for the 3 axes
  short temp;
  while(1) {
    // read the accelerometer from all three axes
    // the accelerometer and the pic32 are both little endian by default (the lowest address has the LSB)
    // the accelerations are 16-bit twos compliment numbers, the same as a short
    acc_read_register(OUT_X_L_A, (unsigned char *)accels,6);

    // NOTE: the accelerometer reads DC accelerations
    // (meaning that, on earth, when stationary it measures gravity as a 1 g acceleration)
    // You could use this information to calibrate the readings into actual units
    sprintf(buffer,"x: %d y: %d z: %d\r\n",accels[0], accels[1], accels[2]);
    NU32_WriteUART1(buffer);

    // need to read all 6 bytes in one transaction to get an update. e
    acc_read_register(OUT_X_L_M, (unsigned char *)mags, 6);

    sprintf(buffer, "xmag: %d ymag: %d zmag: %d \r\n",mags[0], mags[1], mags[2]);
    NU32_WriteUART1(buffer);

    // read the temperature data.  Its a right justified 12 bit two's compliment number
    acc_read_register(TEMP_OUT_L,(unsigned char *)&temp,2);
    sprintf(buffer,"temp: %d\r\n",temp);
    NU32_WriteUART1(buffer);

    //delay
   _CP0_SET_COUNT(0);
   while(_CP0_GET_COUNT() < 40000000) { ; }
  }
}
```

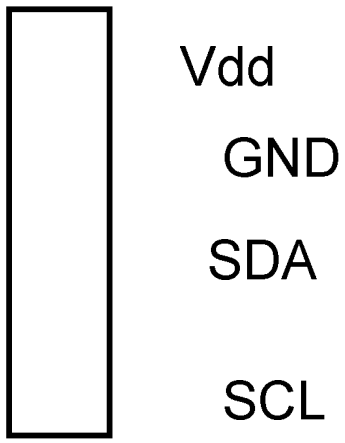
Connect the board as per Fig. 13.3. When used with the Polulu breakout board, the accelerometer has an I²C address of 0x1D. The example code continuously displays the raw accelerometer, magnetometer, and temperature sensor values. Note that the accelerometer measures DC accelerations, so the direction pointed in the direction of the gravitational force will have a large magnitude.

13.3.4 OLED Screen

An organic light emitting diode (OLED) screen is a low-power, high-resolution display. The SSD1603 controller interfaces with an array of OLEDs to control the screen, and the PIC32 uses I²C to communicate with such a chip.

Code Sample 13.12. `i2c_display.h` Header file for controlling an OLED display.

```
#ifndef I2C_DISPLAY_H__
#define I2C_DISPLAY_H__
```



This shows a circuit diagram, illustrating how to wire the LSM303D

Figure 13.3: LSM303D accelerometer connected to the I²C bus.

```
// bare-bones driver for interfacing with the SSD1306 OLED display via I2C
// not fully featured, just demonstrates basic operation
// note that resetting the PIC does not perform a reset on the OLED display, only power cycling does

#define WIDTH 128 //display width in bits
#define HEIGHT 64 //display height, in bits

void display_init(void); // initialize I2C2

void display_command(unsigned char cmd); // issue a command to the display

void display_draw(void);                        // draw the buffer in the display

void display_clear(void);                      // clear the display

void display_pixel_set(int row, int col, int val); // set the pixel at the given row and column

int display_pixel_get(int row, int col);  // get the pixel at the given row and column

#endif
```

Code Sample 13.13. `i2c_display.c` OLED screen interfacing code

```
#include "i2c_master_int.h"
#include "i2c_display.h"
#include <stdlib.h>
// control the SSD1306 OLED display
// Not meant as a full driver, just demonstrates the basics
// note that resetting the PIC does not perform a reset on the OLED display, only power cycling does

#define DISPLAY_ADDR 0x3C

#define SIZE WIDTH*HEIGHT/8 //display size, in bytes

static unsigned char video_buffer[SIZE+1] = {0};  // buffer corresponding to display pixels,
                                                  //for sending over i2c. The first byte allows us to
                                                  // store the control character
```

```
static unsigned char * gddram = video_buffer + 1; // the video buffer start, excluding address byte.
                                                  // we write these pixels to gddram over i2c


void display_command(unsigned char cmd) { // write a command to the display
  unsigned char to_write[] = {0x00,cmd};  // first byte is 0 (CO = 0, DC = 0), second byte is the command
  i2c_write_read(DISPLAY_ADDR, to_write,2, NULL, 0);
}

void display_init() {
  i2c_master_setup();
                          // goes through the reset procedure
  display_command(0xAE);  // turn off display

  display_command(0xA8);     // set the multiplex ratio (how many rows are updated
                             // per oled driver clock) to the number of rows in the display
  display_command(HEIGHT-1); // the multiplex ratio set is the value sent + 1, so subtract 1

                         // we will always write the full display on a single update.
  display_command(0x20); // set address mode
  display_command(0x00); // horizontal address mode
  display_command(0x21); // set column address
  display_command(0x00); // start at 0
  display_command(0xFF); // end at 127
                         // with this address mode, the address will go through all
                         // the pixels and then return to the start,
                         // hence we never need to set the address again

  display_command(0x8d); // charge pump
  display_command(0x14); // turn charge pump on (creates the ~7 Volts needed to light pixels)
  display_command(0xAF); // turn on the display
  video_buffer[0] = 0x40;// co = 0, dc =1, allows us to send data directly from video buffer,
                         //0x40 is the "next bytes have data" byte
}

void display_draw() {    // copies data to the gddram on the oled chip
  i2c_write_read(DISPLAY_ADDR, video_buffer, SIZE + 1, NULL, 0);
}

void display_clear() {
  memset(gddram,0,SIZE);
}

static inline int pixel_pos(int row, int col) { // get the position in gddram of the pixel position
  return (row/8)*WIDTH + col;
}

static inline unsigned char pixel_mask(int row) { // get a bitmask for the actual pixel position, based on row
  return 1 << (row % 8);
}

void display_pixel_set(int row, int col,int val) { // invert the pixel at the given row and column
  if(val) {
    gddram[pixel_pos(row,col)] |= pixel_mask(row);   // set the pixel
  } else {
    gddram[pixel_pos(row,col)] &= ~pixel_mask(row);  // clear the pixel
  }
}
```

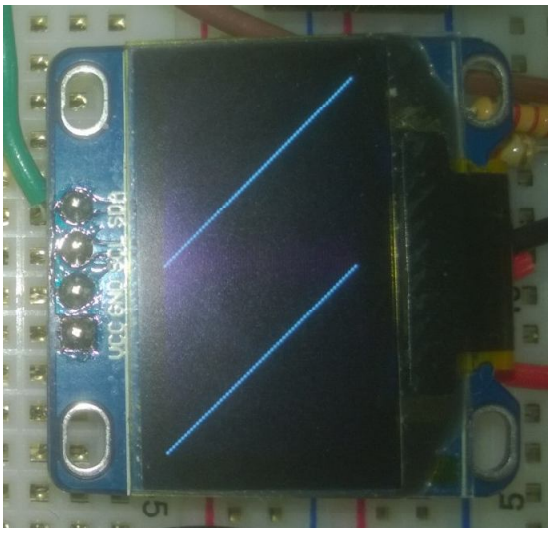


Figure 13.4: An OLED screen

```
int display_pixel_get(int row, int col) {
  return (gddram[pixel_pos(row,col)] & pixel_mask(row)) != 0;
}
```

The sample code provides a simple interface to the OLED display; however, it does not provide comprehensive access to all of the controller's functions. The screen is assumed to be 128 x 64 pixels (that is it has 128 columns and 64 rows). Each pixel is represented by a single bit. The PIC32 stores pixel data in a framebuffer, a copy of the OLED controller's RAM. The functions `display_pixel_set` and `display_pixel_get` access this framebuffer rather than directly accessing the OLED controller's memory. The function `display_draw` copies the whole framebuffer to the OLED controller, which updates the screen.

In the next example, we use the functions in `i2c_display` to draw diagonal lines across the screen.

Code Sample 13.14. `i2c_pixels.c` Draw some lines on an OLED screen.

```
#include "i2c_display.h"
#include "NU32.h"
// Tests the OLED driver by drawing pixels

int main() {
  NU32_Startup();
  display_init();
  int row, col;
  for(col = 0; col < WIDTH; ++col) { // draw a diagonal line
    row = col % HEIGHT;              // when we hit the last row
    display_pixel_set(row,col,1);    // start from row 0, but keep advancing
                                     // the column
    display_draw();                  // we draw every update, to display progress.
  }
  display_draw();

  return 0;
}
```

13.3.5 Multiple Devices

To fully test the OLED screen and accelerometer, we needed to implement a snake game. This example demonstrates how multiple I²C devices can share the same bus: the OLED screen and the accelerometer have different slave addresses so you can connect both of them to the same I²C port. Tilt the device to move the snake in the given direction. Attempt to eat as many dots as possible, and don't crash into yourself or the wall! The code relies on `i2c_accel.c`, `accel.h`, `i2c_master_int.c` and `i2c_master_int.h`.

Code Sample 13.15. `i2c_snake.c` The game of snake, on an OLED screen.

```
#include "i2c_display.h"
#include "accel.h"
#include "NU32.h"

// the game of snake, on an oled display. eat those pixels.

#define MAX_LEN WIDTH*HEIGHT

typedef struct {
  int head;
  int tail;
  int rows[MAX_LEN];
  int cols[MAX_LEN];
} snake_t; // hold the snake

typedef enum {NORTH = 0, EAST = 1, SOUTH = 2, WEST= 3} direction_t; // the direction of the snake

int snake_grow(snake_t * snake, direction_t dir) { // grow the snake in the appropriate direction
  int hrow = snake->rows[snake->head];             // returns false if snake has crashed
  int hcol = snake->cols[snake->head];

  ++snake->head;
  if(snake->head == MAX_LEN) {
    snake->head = 0;
  }
  switch(dir) {                 // move the snake in the appropriate direction
    case NORTH:
      snake->rows[snake->head] = hrow -1;
      snake->cols[snake->head] = hcol;
      break;
    case SOUTH:
      snake->rows[snake->head] = hrow + 1;
      snake->cols[snake->head] = hcol;
      break;
    case EAST:
      snake->rows[snake->head] = hrow;
      snake->cols[snake->head] = hcol + 1;
      break;
    case WEST:
      snake->rows[snake->head] = hrow;
      snake->cols[snake->head] = hcol -1;
      break;
   }
  // check for collisions with the wall or with itself and return 0, otherwise return 1
  if(snake->rows[snake->head] < 0 || snake->rows[snake->head] >= HEIGHT
      || snake->cols[snake->head] < 0 || snake->cols[snake->head] >= WIDTH) {
    return 0;
  } else if(display_pixel_get(snake->rows[snake->head],snake->cols[snake->head]) == 1) {
```

```
      return 0;
  } else {
    display_pixel_set(snake->rows[snake->head],snake->cols[snake->head],1);
    return 1;
  }

}

void snake_move(snake_t * snake)  { // move the snake by deleting the tail
  display_pixel_set(snake->rows[snake->tail],snake->cols[snake->tail],0);
  ++snake->tail;
  if(snake->tail == MAX_LEN) {
    snake->tail = 0;
  }
}


int main() {
  NU32_Startup();
  display_init();
  acc_setup();

  while(1) {
    snake_t snake = {5, 0, {20,20,20,20,20,20},{20,21,22,23,24,25}};
    int dead = 0;
    direction_t dir = EAST;
    char dir_key = 0;
    char buffer[3];
    int i;
    int crow, ccol;
    int eaten = 1;
    int grow = 0;
    short acc[2]; // x and y accleration
    short mag;
    for(i = snake.tail; i <= snake.head; ++i) { // draw the initial snake
      display_pixel_set(snake.rows[i],snake.cols[i],1);
    }
    display_draw();
    acc_read_register(OUT_X_L_M,(unsigned char *)&mag,2);
    srand(mag); // seed the random number generator with the magnetic field
                // (not the most random but if you are moving the PIC a lot it should be good enough for a game)
    while(!dead) {
      if(eaten) {
        crow = rand() % HEIGHT;
        ccol = rand() % WIDTH;
        display_pixel_set(crow,ccol,1);
        eaten = 0;
      }

      //determine direction based on accelerometer, based on the largest magnitude accel and its direction
      acc_read_register(OUT_X_L_A,(unsigned char *)&acc,4);
      if(abs(acc[0]) > abs(acc[1])) { // move the snake in the direction of largest acceleration
        if(acc[0] > 0) {              // prevent the snake from turning 180 degress, resulting
          if(dir != EAST) {           // in an automatic self crash
            dir = WEST;
          }
        } else
          if( dir != WEST) {
            dir = EAST;
```

```
          }
      } else {
        if(acc[1] > 0) {
          if( dir != SOUTH) {
            dir = NORTH;
          }
        } else {
          if( dir != NORTH) {
            dir = SOUTH;
          }
        }
      }


      if(snake_grow(&snake,dir)) {
        snake_move(&snake);
      } else if(snake.rows[snake.head] == crow && snake.cols[snake.head] == ccol) {
          eaten = 1;
          grow += 15;
      } else {
        dead = 1;
        display_clear();
      }
      if(grow > 0) {
          snake_grow(&snake,dir);
          --grow;
      }
      display_draw();
    }
  }



  return 0;
}
```

13.4 Chapter Summary

- I²C communication requires two wires; one for the clock (SCL) and another for data (SDA). Both lines should be pulled up to 3.3V with resistors.
- I²C devices can be either a master or a slave. The master controls the clock and initiates all communication. Usually the PIC32 operates as the master, but it can also be a slave.
- Each slave has an address, and only responds when the master issues its address, allowing multiple slaves to be connected to a single master. Multiple masters can also coexist on the same wire, but only one master can operate at a time.
- The I²C peripheral automatically handles the low-level components of I²C communication. Using the peripheral, you can issue primitive commands. A full I²C transaction, however, requires a sequence of these basic primitive commands that must be handled in software. The master can sequence the commands by polling for certain status flags or by using an ISR.

Chapter 14

Parallel Master Port

The parallel master port (PMP) provides a mechanism for communicating with devices in parallel. Rather than sending each bit sequentially, as in serial communication, parallel communication uses multiple wires to send several bits simultaneously. Although parallel communication allows transferring more bits per clock cycle, synchronization and electrical interference severely limit the frequency of parallel protocols. Nevertheless, parallel communication provides a simple interface to some devices such some LCD controllers.

14.1 Overview

The PIC32 has one PMP peripheral. Despite its name, the PMP can act as a master or a slave; however, we focus on using it as a master. As master, the PMP has different modes of communicating with the slave, making it suitable for interfacing with various of parallel devices.

The PIC32's PMP can use up to 16 address pins, 16 data pins, and an assortment of control pins as well. The address pins contain location information for the device, determining which remote memory address to read to or write from. The peripheral can optionally increment the address after each read/write to enable reading from/writing to multiple registers on the target device. The data pins are used for sending bytes between the PMP and the device. Control pins implement hardware handshaking, a signaling mechanism that tells both the PIC32 and device when data is available on the pins and the direction of the data transfer (read or write).

The PMP supports two basic methods of hardware handshaking; master modes 1 and 2. Both methods involve two pins and pulsed signals called strobes. A strobe is a single pulse used to signal the slave device; it also refers to the pin that issues the pulse. In master mode 2, the pins are called parallel master read (PMRD) and parallel master write (PMWR). To read from the device, the PMP pulses the read strobe (PMRD), signaling the device to output a byte to the data pins. Writing requires pulsing the write strobe (PMWR), which signals the device to read a byte from the data pins. Master mode 1 uses a signal called (PMRD/PMWR) and a strobe called parallel master enable. The PMRD/PMWR signal determines whether the operation is a read or a write while PMENB issues the strobe signal. The hardware handshaking process has various timing requirements, depending on the slave. The PMP can be configured with wait states, which determine how long the data must be valid on the pins before and after the strobe pulse, as well as the duration of the pulse itself. See Fig. 14.1 for timing details.

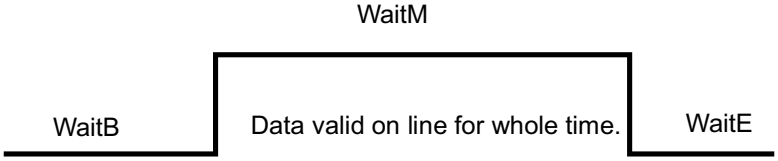


Figure 14.1: Timing for parallel communication.

14.2 Details

The parallel master port uses several SFRs; however, some are not used in master mode. We omit the SFRs used only in slave mode.

PMCON The main configuration register.

PMCON⟨15⟩ or PMCONbits.ON: Setting this bit enables the PMP.

PMCON⟨9⟩ or PMCONbits.PTWREN: Setting this bit enables the PMWR/PMENB pin. The pin is called PMWR when using master mode 2 and PMENB when using master mode 1. Most devices require the use of PMWR/PMENB so you will usually set this bit.

PMCON⟨8⟩ or PMCONbits.PTRDEN: Setting this bit enables the PMRD/PMWR pin. The pin is called PMRD when using master mode 2 and PMRD/PMWR when using master mode 1. Unless you are using dual strobe mode and do not need to read from the slave you should set this bit.

PMCON⟨1⟩ or PMCONbits.WRSP: Determines the polarity of either PMWR or PMENB, depending on the mode, master mode 2 or master mode 1 respectively. When set, the pin is active high.

PMCON⟨0⟩ or PMCONbits.RDSP: Determines the polarity of either PMRD or PMRD/PMWR depending on the mode, master mode 2 or master mode 1 respectively. When set, the pin is active high.

PMMODE Controls the PMP's operating mode and tracks its status.

PMCON⟨15⟩ or PMCONbits.BUSY: The PMP sets this bit when the peripheral is busy. Poll this bit to determine when a read or write operation has finished.

PMCON⟨9:8⟩ or PMCONbits.MODE: Determines the mode of the PMP. The two master modes are

0b11 Master mode 1, uses a single strobe PMENB and a read/write direction signal PMRD/PMWR.

0b10 Master mode 2, uses a read strobe PMRD and a write strobe PMWR.

PMCON⟨7:6⟩ or PMCONbits.WAITB: Determines the amount of time between initiating a read/write and triggering the strobe. The time inserted will be $(1 + \text{PMCONbits.WAITB})T_{pb}$, where T_{pb} is the peripheral bus clock period, for a minimum of one wait state (T_{pb}) and a maximum of four wait states ($4T_{pb}$). See Fig. 14.1.

PMCON⟨5:2⟩ or PMCONbits.WAITM: Determines the duration of the strobe. The duration is given by $(1 + \text{PMCONbits.WAITM})T_{pb}$, for a minimum of one wait state (T_{pb}) and a maximum of 16 wait states ($16T_{pb}$). See Fig. 14.1

PMCON⟨1:0⟩ or PMCONbits.WAITE: Determines how long to wait after the end of the strobe. For reads, the number of wait states is PMCONbits.WAITE and for writes it is PMCONbits.WAITE + 1. Each wait state is one T_{pb} long. See Fig. 14.1.

PMADDR Set this register to the desired read/write address before performing the operation. The peripheral may also be configured to automatically increment this register after each read or write, allowing you to access a series of consecutive registers on the slave device.

PMDIN In either master mode, use this register to read from and write to the slave. Writes to this register are sent to the slave. Reading a value from the PMP requires you to read from this SFR twice. The first read initiates a read sequence on the bus and stores the value in an internal buffer. The subsequent read of this register returns the actual value of the read operation. Hence, the value read from PMDIN always lags the latest value returned from the slave by one read.

PMAEN Determines which parallel port address pins are used by the port and which are controlled by other peripherals. To claim a pin for the PMP set the corresponding bit to 1. All the PMP address port pins are labeled PMA0 - PMA15. Some of these pins can serve multiple purposes for the PMP port, but we have not delved into such details here.

14.3 Sample Code

The sample code below is the implementation of the LCD library for a Hitachi HD44780 or compatible LCD controller. The header file may look familiar to you, as it is the one used in Ch. 4. The functions that interact directly with the PMP are `wait_pmp`, `LCD_Setup`, `LCD_Read`, and `LCD_Write`. Notice that `LCD_Read` actually reads data twice; the first read initiates the reading sequence and stores data in the PMP buffer, while the second read returns the data from the buffer.

Code Sample 14.1. `LCD.h` An LCD control library using the PMP.

```
#ifndef LCD_H
#define LCD_H
// LCD control library for Hitachi HD44780 compatible LCDs.

void LCD_Setup(void);                       // Initialize the LCD
void LCD_Clear(void);                       // Clear the screen and return to position (0,0)
void LCD_Move(int line, int col);           // Move the position to the given line and column
void LCD_WriteChar(char c);                 // Write a character at the current position
void LCD_WriteString(const char * string);  // Write a string, starting at the current position

void LCD_Home(void);                        // Move to (0,0) and reset any scrolling
void LCD_Entry(int id, int s);              // Control how the display moves after sending a character
void LCD_Display(int d, int c, int b);      // Turn the display on/off and change cursor settings
void LCD_Shift(int sc, int rl);             // Shift the position of the display
void LCD_Function(int n, int f);            // Set the number of lines (0,1) and the font size
void LCD_CustomChar(unsigned char val, const char data[7]); // Write a custom character to CGRAM
void LCD_Write(int rs, unsigned char db70); // Write a command to the LCD
void LCD_CMove(unsigned char addr);         // Move to the given address in CGRAM
unsigned char LCD_Read(int rs);             // Read a value from the LCD
#endif
```

Code Sample 14.2. `LCD.c` Implementation of an LCD control library using the PMP.

```
#include "LCD.h"
#include<xc.h>

// wait for the peripheral master port (PMP) to be ready
// should be called before every read and write operation
static void waitPMP(void)
{
  while(PMMODEbits.BUSY) {
    ;
  }
}


// wait for the lcd to finish its command.
// We check this by reading from the LCD
static void waitLCD() {
  volatile unsigned char val = 0x80;

  // Read from the LCD until the Busy flag (BF, 7th bit) is 0,
  while (val & 0x80) {
    val = LCD_Read(0);
  }
  int i = 0;
```

```
  for(i = 0; i < 50; ++i) {
    _nop();
  }
}


// setup the parallel master port (PMP) to control the LCD
// pins RE0 - RE7 (PMD0 - PMD7) connect to LCD pins D0 - D7
// pin RD4 (PMENB) connects to LCD pin E
// pin RD5 (PMRD/PMWR) Connects to LCD pin R/W
// pin RD11 (PMA14) Connects to RS.
// interrupts will be disabled while this function executes
void LCD_Setup() {
  int en = __builtin_disable_interrupts();  // disable interrupts

  IEC1bits.PMPIE = 0;    // disable PMP interrupts
  PMCON = 0;             // clear PMCON, like it is on reset
  PMCONbits.PTWREN = 1;  // PMENB strobe enabled
  PMCONbits.PTRDEN = 1;  // PMRD/PMWR enabled
  PMCONbits.WRSP = 1;    // Read/write strobe is active high
  PMCONbits.RDSP = 1;    // Read/write strobe is active high

  PMMODE = 0;            // clear PMMODE like it is on reset
  PMMODEbits.MODE = 0x3; // set master mode 1, which uses a single strobe

  // Set up wait states.  The LCD requires data to be held on its lines
  // for a minimum amount of time before it works
  // All wait states are given in peripheral bus clock
  // (PBCLK) cycles.  PBCLK of 80 Mhz in our case
  // so one cycle is 1/80Mhz = 12.5ns.
  // The timing controls asserting/clearing PMENB (RD4) which
  // is connected to the E pin of the LCD (we refer to the signal as E here)
  // The times required to wait can be found in the LCD controller's data sheet.
  // The cycle is started when reading from or writing to the PMDIN SFR.
  // Note that the wait states for writes start at 1 (except WAITE)
  // We add some extra wait states to make sure we meet the time and
  // account for variations in timing amongst different HD4780 compatible parts.
  // The timing we use here is for the KS066U which is faster than the HD4780
  PMMODEbits.WAITB = 0x3;  // Tas in the LCD datasheet is 60 ns
  PMMODEbits.WAITM = 0xF;  // PWeh in the data sheet is 230 ns (we don't meet this but do our best)
                           // If not working for your LCD you may need to reduce PBCLK
  PMMODEbits.WAITE = 0x1;  // after E is low wait Tah (10ns)

  PMAENbits.PTEN14 = 1; // PMA14 is an adddress port

  PMCONbits.ON = 1;      // enable the PMP peripheral
  // perform the initialization sequence
  LCD_Function(1,0);     // 2 line mode, small font
  LCD_Display(1, 0, 0);  // Display ON/OFF control: display on cursor off blinking cursor off
  LCD_Clear();          // clear the LCD
  LCD_Entry(1, 0);       // Cursor moves left to right. do not shift the display

  if(en & 0x1)          // if interrupts were enabled when calling this function, enable them
  {
    __builtin_enable_interrupts();
  }
}

// Clears the display and return to the home position (0,0)
```

```
void LCD_Clear(void) {
  LCD_Write(0,0x01); //clear the whole screen
}

// Return the cursor and display to the home position (0,0)
void LCD_Home(void) {
  LCD_Write(0,0x02);
}

// Issue the LCD entry mode set command
// This telss the LCD what to do after writing a character
// id : 1 increment cursor, 0 decrement cursor
// s : 1 shift display right, 0 don't shift display
void LCD_Entry(int id, int s) {
  LCD_Write(0, 0x04 | (id << 1) | s);
}

// Issue the LCD Display command
// Changes display settings
// d : 1 display on, 0 display off
// c : 1 cursor on, 0 cursor off
// b : 1 cursor blinks, 0 cursor doesn't blink
void LCD_Display(int d, int c, int b) {
  LCD_Write(0, 0x08 | (d << 2) | (c << 1) | b);
}

// Issue the LCD display shift command
// Move the cursor or the display right or left
// sc : 0 shift cursor, 1 shift display
// rl : 0 to the left, 1 to the right
void LCD_Shift(int sc, int rl) {
  LCD_Write(0,0x1 | (sc << 3) | (rl << 2));
}

// Issue the LCD Functions set command
// This controls some LCD settings
// You may want to clear the screen after calling this
// n : 0 one line, 1 two lines
// f : 0 small font, 1 large font (only if n == 0)
void LCD_Function(int n, int f) {
  LCD_Write(0, 0x30 | (n << 3) | (f << 2));
}

// Move the cursor to the desired line and column
// Does this by issuing a DDRAM Move instruction
// line : line 0 or line 1
// col  : the desired column
void LCD_Move(int line, int col) {
  LCD_Write(0, 0x80 | (line << 6) | col);
}

// Sets the CGRAM address, used for creating custom
// characters
// addr address in the CGRAM to make current
void LCD_CMove(unsigned char addr) {
  LCD_Write(0, 0x40 | addr);
}

// Writes the character to the LCD at the current position
```

```
void LCD_WriteChar(char c) {
  LCD_Write(1, c);
}

// Write a string to the LCD starting at the current cursor
void LCD_WriteString(const char *str) {
  while(*str) {
    LCD_WriteChar(*str); // increment string pointer after char sent
    ++str;
  }
}

// Make val a custom character.  This only implements
// The small font version
// val : between 0 and 7
// data : 7 character array.  The first 5 bits of each character
//        determine whether that pixel is on or off
void LCD_CustomChar(unsigned char val, const char * data) {
  int i = 0;
  for(i = 0; i < 7; ++i) {
    LCD_CMove(((val & 7) << 2) | i);
    LCD_Write(1, data[i]);
  }
}

// Write data to the LCD and wait for it to finish by checking the busy flag.
// rs : the value of the RS signal, 0 for an instruction 1 for data
// data : the byte to send
void LCD_Write(int rs, unsigned char data) {
  waitLCD();           // wait for the LCD to be ready
  PMADDRbits.CS1 = rs; // 0 for command, 1 for data
  waitPMP();           // Wait for the PMP to be ready
  PMDIN = data;        // send the data
}

// read data from the LCD.
// rs : the value of the RS signal 0 for instructions status, 1 for data
unsigned char LCD_Read(int rs) {
  volatile unsigned char val = 0; // volatile so the first read statement does not get optimized away
  PMADDRbits.CS1 = rs;        // low to read command status, high to read data
  // from the PIC32 reference manual, you must read twice to actually get the data
  waitPMP();                  // wait for the PMP to be ready
  val = PMDIN;
  waitPMP();
  val = PMDIN;
  return val;
}
```

14.4 Chapter Summary

- The PMP sends multiple bits of data simultaneously; however, this requires using many more wires than serial communication protocols.
- The PMP performs hardware handshaking automatically. This process tells the slave device when to read data and when to output data.

- Devices with parallel ports use slightly different protocols; hence the PMP is highly configurable.

Chapter 15

Input Capture

The input capture (IC) peripheral monitors an external signal and stores a timer value when that signal changes, allowing precise timing of external events. In some sense, input capture can be viewed as the opposite of output compare. Output compare changes the value of an output pin, based on the value of a timer, whereas input capture stores the value of a timer based on the value of an input pin.

15.1 Overview

The PIC32 has five input capture modules, each associated with a single pin. The input capture peripheral monitors the voltage on the pin and can trigger on external events such as a rising edge or falling edge of the signal. When the specified event occurs, the module stores the value of a timer in a FIFO buffer and, optionally, triggers an interrupt. Thus, each event receives a time-stamp from the PIC32, allowing software to know when a certain event occurred without needing to immediately respond to it in software. Figure 15.1 depicts the operation of the IC module.

Prior to using the IC module, you should configure a timer. The frequency of the timer determines the precision of the times captured by the IC peripheral. The IC peripheral always synchronizes the input event with the system clock; therefore, if the timer runs at the peripheral bus clock frequency there may be up to a three cycle delay between when the event occurs and when timer value is recorded. Slower timers, however, do not have this issue because the data can be synchronized within one timer period.

15.2 Details

The input capture peripheral uses two SFRs, ICxCON and ICxBUF.

ICxCON The main IC control register.

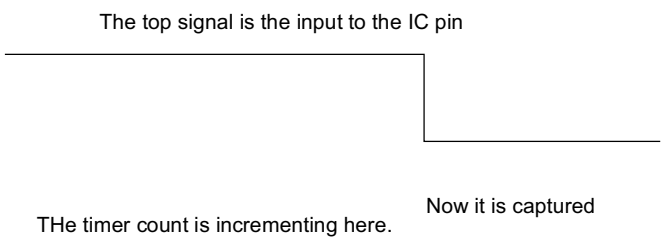


Figure 15.1: Timing of the input capture module. The falling edge of the input triggers the capture event.

ICxCON⟨15⟩ or ICxCONbits.ON: Set to one to enable the module. Clear to zero to disable the module and reset it.

ICxCON⟨9⟩ or ICxCONbits.FEDGE: Determines whether, in the applicable mode, the first edge captured is a rising edge (1) or falling edge (0).

ICxCON⟨8⟩ or ICxCONbits.C32: If one, the IC peripheral uses the 32-bit timer Timer23. If zero, the IC uses a 16-bit timer.

ICxCON⟨7⟩ or ICxCONbits.ICTMR: Determines which timer to use, if ICxCONbits.C32 is zero. If ICxCONbits.C32 is one, this value is ignored

0 Use Timer3.

1 Use Timer2

ICxCON⟨6:5⟩ or ICxCONbits.ICI: Interrupt after $ICI + 1$ capture events.

ICxCON⟨4⟩ or ICxCONbits.ICOV: Input capture overrun, read-only. The IC peripheral has a FIFO buffer that can store four timer values. When this FIFO is full and another capture event occurs hardware sets ICxCONbits.ICOV. In certain modes, ICxCONbits.ICOV will not be set even when the buffer is full. To clear ICxCONbits.ICOV you must read from the ICxBUF, removing the overflow condition.

ICxCON⟨3⟩ or ICxCONbits.ICBNE: Read-only. Set whenever event times are available in the input capture FIFO, accessed via ICxBUF.

ICxCON⟨2:0⟩ or ICxCONbits.ICM: Determines the events that cause the IC module to trigger.

0b111 Allows the IC pin to be used to wake the PIC32 from sleep.

0b110 First trigger on the edge specified by ICxCONbits.FEDGE. Afterwards, trigger on every edge.

0b101 Capture every sixteenth rising edge.

0b100 Capture every fourth rising edge.

0b011 Capture every rising edge.

0b010 Capture every falling edge.

0b001 Capture ever edge.

0b000 Input Capture disabled.

ICxBUF Read-only buffer that returns the timer value captured by the IC peripheral. This SFR reads the next element from a 4-value deep FIFO; thus, four input capture events can be queued at one time. When the FIFO contains data, ICxCONbits.ICBNE is set, indicating that a read from ICxBUF will contain a captured time-stamp. To clear the overflow flag, ICxCONbits.ICOV, read from ICxBUF, which removes a value from the FIFO.

15.3 Sample Code

The following code uses the input capture to measure PWM duration. First it configures Timer2 for 10 MHz operation. Next it configures the input capture to trigger on the falling edge and read the value from . We use an interrupt to actually read the captured value. Finally, output compare one is configured to output a PWM signal. The user is prompted for a duty cycle, which is then measured by the input capture and reported.

Code Sample 15.1. `input_capture.c` Basic input capture example.

```
#include "NU32.h"
// use input capture 1 (RD8) to measure the PWM duty cycle of OC1 (D0)
// Connect RD8 to D0 for this to work
// Alternatively, you may connect an external PWM signal to RD8 and use this program
// to measure it (without modification)
```

```
volatile int changed = 0;
volatile int ic_dcycle = -1; // duty cycle, as measured by IC1
void __ISR(_INPUT_CAPTURE_1_VECTOR, IPL3SOFT) InputCapture1() {
  int dcycle = IC1BUF;
  if(dcycle != ic_dcycle) {
    ic_dcycle = dcycle;
    changed = 1;
  }

  IFS0bits.IC1IF = 0;
}


int main() {
  char buffer[100]="";
  NU32_Startup();
  __builtin_disable_interrupts();


                        // set up timer2 with a 1:8 prescaler (10 Mhz operation)
  T2CONbits.TCKPS = 0x3;// 1:8 prescaler; ticks at 10 Mhz (each tick is 100 ns)
  PR2 = 9999;           // rollover after 1 second
  TMR2 = 0;
                        // set up IC1 to trigger on both rising and falling edges and to use a 16 bit clock
                        // to check for timer rollover


                        // set up the input capture
  IC1CONbits.ICM = 2;   // capture on the falling edge
  IC1CONbits.ICTMR = 1; // use timer 2
  IFS0bits.IC1IF = 0;
  IPC1bits.IC1IP = 3;   // interrupt priority 3
  IEC0bits.IC1IE = 1;   // enable the input capture input

  OC1CONbits.OCM = 0b110;   // PWM mode without fault pin; other OC1CON bits are defaults
                            //(OC1 uses timer 2 as well)
  OC1RS = 2500;             // duty cycle = OC1RS/(PR2+1) = 25%
  OC1R = 2500;              // initialize before turning OC1 on; afterward it is read-only

  T2CONbits.ON = 1;        // turn on Timer2
  OC1CONbits.ON = 1;       // turn on OC1
  IC1CONbits.ON = 1;       // turn on IC1

  __builtin_enable_interrupts();

  while(1) {
    int val;
    NU32_WriteUART1("Enter duty cycle (0-9999)\r\n");
    NU32_ReadUART1(buffer,100);
    sscanf(buffer,"%d",&val);
    OC1RS = val;
    while(!changed) {
      ; // wait for input capture to recognize the change
    }
    sprintf(buffer,"Duty Cycle changed to %d\r\n",ic_dcycle);
    NU32_WriteUART1(buffer);
    changed = 0;
  }
}
```

15.4 Chapter Summary

- The input capture peripheral allows you to record the time that an external event occurs.
- The input capture peripheral can be configured to trigger on rising edges, falling edges, or combinations thereof.
- The precision of the time recorded depends on the frequency of the timer used. When the timer runs at the peripheral bus clock frequency there may be a few cycle lag between when an event occurs and the timer value that is recorded.

Chapter 16

Comparator

Comparators compare two analog voltages, outputting a digital signal depending on their relative magnitudes. Thus, comparators function as 1-bit analog to digital converters. Figure 16.1 depicts a comparator. If the voltage at the non-inverting ("+") input is larger than than the voltage at the inverting ("-") input the output voltage will be high (1); otherwise it will be low (0).

16.1 Overview

The PIC32 has two comparators. Each comparator can compare voltages internally generated by the PIC32 and voltages supplied by external pins. Comparator output can be read from the PIC32, used to trigger interrupts, or output to an external pin. Additionally, the internal comparator reference voltage can be output to an external pin, which results in a cheap digital to analog converter (DAC). An internal resistor network, depicted in 16.2 provides two distinct output ranges with 16 voltage values each. Typically, you need to buffer the output from the internal reference voltage. Although the comparators and internal reference voltage generation are intimately related, they are discussed in separate reference manual chapters.

16.2 Details

Each comparator has a single control SFR. They both share a status SFR.

CMxCON Control register for the comparators.

CMxCON⟨15⟩ or CMxCONbits.ON: Set to one to enable the comparator.

CMxCON⟨14⟩ or CMxCONbits.COE: Setting to one causes the comparator's output to drive the CxOUT pin. When set, the corresponding TRIS SFR bit should be cleared to zero, making it an output (0). C1OUT corresponds to pin RB8 and C2OUT is pin RB9.

CMxCON⟨13⟩ or CMxCONbits.CPOL: When set, inverts the comparator's output.

CMxCON⟨8⟩ or CMxCONbits.COUT: The output of the comparator. Can also be read from CMSTAT.

CMxCON⟨7:6⟩ or CMxCONbits.EVPOL: Controls the conditions under which the comparator generates an interrupt.

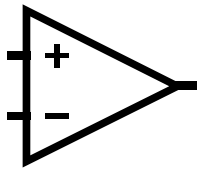


Figure 16.1: Comparator

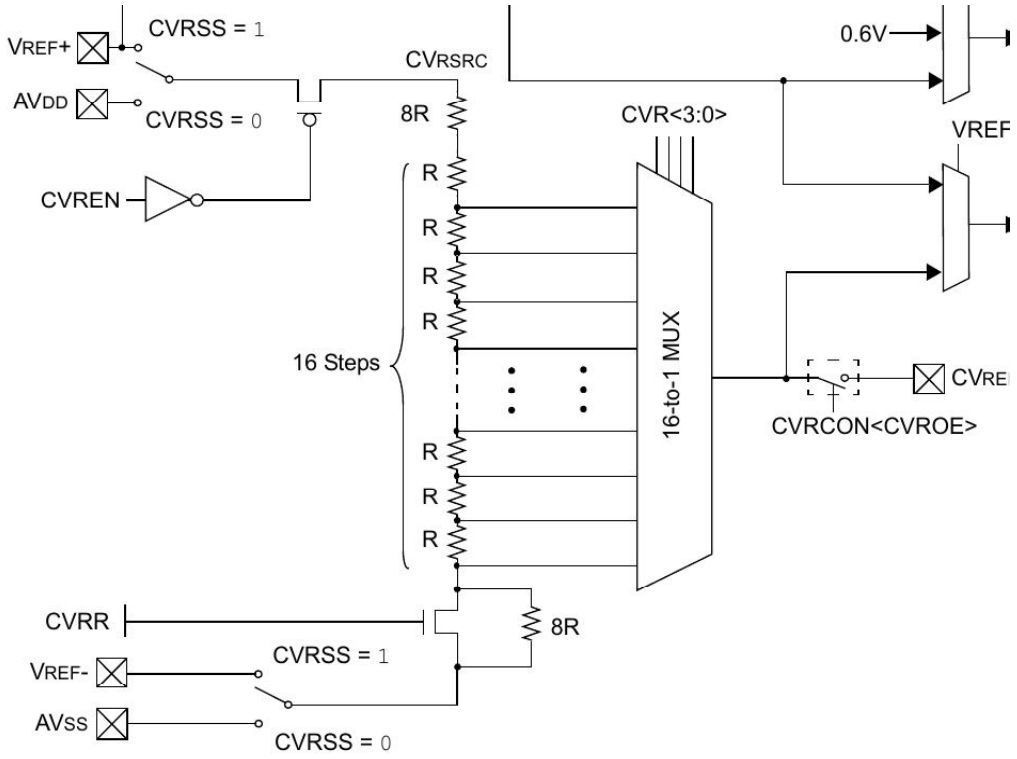


Figure 16.2: The internal resistor network that creates the reference voltage

0b11 Interrupt when the comparator output transitions from high to low or low to high.

0b10 Interrupt when the comparator output transitions from high to low.

0b01 Interrupt when the comparator output transitions from low to high.

0b00 Do not interrupt.

CMxCON⟨4⟩ or CMXCONbits.CREF: Determines what is connected to the non-inverting (+) comparator input.

1 Internal CV_{REF}.

0 External CxIN+ pin. If you are using USB, then C1IN+ cannot be used as an input pin, so this field must be set to 1.

CMxCON⟨1:0⟩ or CMxCONbits.CCH: Controls what is connected to the inverting (-) comparator input. Let y represent the alternative comparator number; that is, using CM1CON (x = 1), y = 2 and if using CM2CON (x =2) y = 1. Then the meaning of the CMxCONbits.CCH bit field is determined as follows:

0b11 Connected to IV_{REF}, the internal voltage reference, which is nominally 1.2 V.

0b10 External CyIN+ pin.

0b01 External CxIN+ pin.

0b00 External CxIN- pin.

CMSTAT The comparator status register, shared by both comparators.

CMSTAT⟨1⟩ or CMSTATbits.C2OUT: The output of comparator 2.

CMSTAT⟨0⟩ or CMSTATbits.C1OUT: The output of comparator 1.

CVRCON The comparator reference voltage control register. Controls the reference voltage value and whether it is output to a pin.

CVRCON⟨15⟩ or CVRCONbits.ON: Set to one to enable the comparator reference voltage.

CVRCON⟨6⟩ or CVRCONbits.CVROE: Set to output the reference voltage on the CVREFOUT (RB9) pin. If clear, voltage will only be accessible internally.

CVRCON⟨5⟩ or CVRCONbits.CVRR: Controls the voltage range.

1 0V-2.06 volts, with a step size of approximately 0.14V, when using the internal ADC voltage references AV_{dd} and AV_{ss}.

0 0.83V-2.37V, with a step size of approximately 0.103V, when using the internal ADC voltage references AV_{dd} and AV_{ss}.

CVRCON⟨4⟩ or CVRCONbits.CVRSS: Selects the source for the comparator voltage reference CV_{RSRC}.

1 : Uses external voltage references $V_{REF}+$ (RA10) and $V_{REF}-$ (RA9), so $CV_{RSRC} = V_{REF}+ - V_{REF}-$.

0 : The default, uses the analog supply voltages AV_{dd} (3.3V) and AV_{ss} (0V), resulting in a CV_{RSRC} of 3.3V.

CVRCON⟨3:0⟩ or CVRCONbits.CVR: Determines the actual voltage output, CV_{REF}. Depends on the settings of CVRCONbits.CVRSS and CVRCONbits.CVRR. When CVRCONbits.CVRR = 1:

$$CV_{REF} = (\text{CVRCONbits.CVR}/24) \times CV_{RSRC} \tag{16.1}$$

and with CVRCONbits.CVRR = 0:

$$CV_{REF} = (0.25 + \text{CVRCONbits.CVR}/32) \times CV_{RSRC} \tag{16.2}$$

16.3 Sample Code

16.3.1 Voltage Comparison

The following code uses comparator 2 to compare an external voltage to an internally generated reference signal. The internal reference voltage is $IV_{REF} = 1.2$ V, and the external voltage is applied to the non-inverting (+) input via C2IN+ (RB3). Whenever the voltage on C2IN+ is greater than 1.2V both LEDs will illuminate. You can test the example by applying the output of various voltage dividers to the C2IN+ pin.

Code Sample 16.1. `comparator.c` Basic comparator example.

```
#include "NU32.h"
// Uses a comparator to interrupt on a low voltage condition
//
// The + comparator terminal is connected to C2IN+ (RB3)
// The - comparator terminal is connected to the internal voltage IVref (1.2V)
// The output of the comparator can be viewed on C2OUT(RB9) with a voltmeter or measured with the PIC32
//
// Attach the outputs of various voltage dividers to C2IN+.
// When The voltage is > 1.2V both LEDS will illuminate
// and when they are < 1.2V both LEDs will be off
int main() {
  NU32_Startup();
                    // configure the comparator
  CM2CONbits.COE = 1; // comparator output is on the C2OUT pin, so you can measure it with a voltmeter
  CM2CONbits.CCH = 0x3; // Connect - output to IVref
  TRISBbits.TRISB9 = 0; // configure B9 as an output, which must be set to use C2OUT
  CM2CONbits.ON = 1;

  while(1) {
    // test the comparator output
    if(CMSTATbits.C2OUT) { // if output is high then the input signal > 1.2V
```

```
      NU32_LED1 = 0;
      NU32_LED2 = 0;
    } else {
      NU32_LED1 = 1;
      NU32_LED2 = 1;
    }
  }
  return 0;
}
```

16.3.2 Analog Output

This example demonstrates how to output the internal comparator reference voltage to a pin. This reference voltage has relatively high output impedance, since it is designed to be input into a comparator; therefore, the output of this reference needs to be output to a buffer in most cases. Without a buffer circuit, you can look at the output voltage using either a voltmeter or an oscilloscope. The reference voltage cycles through the 16 available values, once per second.

Code Sample 16.2. `ref_volt.c` Output comparator reference voltage to a pin.

```
#include "NU32.h"
// Use the comparator reference voltage as a cheap DAC
// Voltage is output on CVREFOUT (RB10). You can measure it.
// You will most likely need to buffer this output to use it to
// drive a low-impedance load.

int main() {
  int i;
  NU32_Startup();
  CVRCONbits.CVROE = 1; // output the voltage on CVrefout
  CVRCONbits.CVRR = 0; // use the smaller range with more voltages (and a higher max voltage)
  TRISBbits.TRISB10 = 0;
  CVRCONbits.ON = 1;   // turn the module on
  while(1) {
    for(i = 0; i < 16; ++i) { // step through the voltages, one per second.
      CVRCONbits.CVR = i;
      _CP0_SET_COUNT(0);
      while(_CP0_GET_COUNT() < 80000000) {
        ;
      }
    }
  }
  return 0;
}
```

16.4 Chapter Summary

- Comparators allow you to compare two analog voltages and determine which one is larger.
- Comparators can compare two external signals or an external signal with an internal signal.
- The comparator reference voltage can be output to an external pin, serving as an imprecise but quick DAC.

Chapter 17

Sleep, Idle, and the Watchdog Timer

In battery powered applications, conserving energy is important. The PIC32 provides multiple power saving modes which help reduce energy consumption. Most of the modes involve using alternative, slower oscillators for either the peripheral or system clocks. Beyond merely changing the clock frequency, in idle mode the CPU can stop executing instructions while the peripherals operate. To save the most power the CPU can enter sleep mode, where only select peripherals operate and the system clock stops entirely. The PIC32 can exit sleep mode in reaction to changes on external pins. Additionally, a special hardware timer, called the watchdog timer (WDT), can also wake the PIC32 without any external signals. The WDT also enables software to recover from certain software defects.

17.1 Overview

17.1.1 Power Saving

The PIC32 has several oscillator sources that can drive the system clock and peripheral clock. Depending on the oscillator source and frequency, these oscillators can consume more or less power. Lower frequencies results in less power consumption but also lower processing speeds. Rather than focus on the numerous oscillator sources and settings, we assume that both the system clock and peripheral bus clock are operating at 80 MHz from the primary oscillator source Posc. Thus, the power saving methods we focus on involve disabling the CPU and peripherals.

Given fixed frequencies, the primary power saving modes for the PIC32 are called sleep and idle. In both modes, the CPU stops executing instructions. In idle mode, however, the system clock continues running and most peripherals can function; therefore, sleep mode conserves more power and idle conserves less power.

To enter sleep or idle mode you must set some bit-fields and then issue the assembly instruction `wait` to actually enter the desired power saving mode. When a peripheral interrupt wakes the PIC32 from sleep, code will continue executing from where the `wait` instruction was issued. The WDT, however, wakes the PIC32 by causing a reset. You can detect when the WDT has reset the PIC32 during sleep or idle mode and then issue an `iret` instruction to resume the code from where the `wait` instruction was issued.

17.1.2 Watchdog Timer

The WDT can not only be used to wake the PIC32 from sleep or idle, but also to aid in recovery from certain software errors. When enabled, the WDT increments until it overflows, resetting the PIC32. To prevent a reset, user software must periodically write to a WDT SFR to reset the timer. If software gets stuck (say in an undesired infinite loop) it will not reset the WDT counter and the PIC32 will reset, providing an escape from the infinite loop.

Unlike other peripherals, the primary setup of the WDT occurs in the configuration bits, set when the device is programmed with an external programmer. For the NU32 board, these bits were set when the bootloader was installed. The configuration bits control whether the WDT is on or off and the timeout period. If enabled in the configuration bits, software cannot disable the WDT; however, software can enable it.

Additionally, software cannot modify the timeout period. Preventing software from changing WDT settings acts as a safety mechanism: buggy software cannot disable the error recovery mechanism that the WDT provides.

17.2 Details

Several SFRs and configuration words control the WDT and power saving modes.

DEVCFG1 The configuration word that contains the WDT configuration. This configuration word can only be changed with a programming device. When the bootloader was loaded onto the NU32, this word was set to disable the watchdog timer and set a period of 4.096 seconds.

DEVCFG1⟨23⟩ or FWDTEN: Watchdog timer enable bit. If one, WDT is enabled and cannot be disabled. The bootloader sets this bit to zero using `#pragma config FWDTEN = OFF`, allowing you to selectively enable or disable the WDT in software.

DEVCFG1⟨20:16⟩ or WDTPS: Watchdog timer postscaler select bits: each setting controls the period of the watchdog timer, according to the data sheet/reference manual. When the bootloader was installed, the programmer set these bits to 0b01100 using `#pragma config WDTPS = PS4096`, which configures a postscaler of 1:4096, corresponding to a WDT period of 4.096 seconds.

WDTCON WDTCON⟨15⟩ or WDTCONbits.ON: Set to enable the watchdog timer. The device configuration overrides this bit, so you cannot disable the WDT if it is enabled in the configuration bits.

WDTCON⟨6:2⟩ or WDTCONbits.SWDTPS: Read-only bits containing the value of the WDT postscaler configuration, DEVCFG1⟨20:16⟩ (WDTPS).

WDTCON⟨0⟩ or WDTCONbits.WDTCLR: Write a 1 to this bit to clear the WDT. If the WDT is enabled and you do not write a 1 to this bit often enough the PIC32 will reset.

RCON RCON⟨4⟩ or RCONbits.WDTO: Set to 1 if the reset has occurred due to a WDT timeout.

RCON⟨3⟩ or RCONbits.SLEEP: Set to 1 if the device had been in sleep mode.

RCON⟨2⟩ or RCONbits.IDLE: Set to 1 if the device had been in idle mode.

OSCCON Allows you to change some oscillator settings.

OSCCON⟨4⟩ or OSCCONbits.SLPEN: When set, the `wait` instruction enters sleep mode. When clear, the `wait` instruction enters idle mode.

SIDLE Stop in IDLE. Not an SFR, but rather a bit-field present in most peripherals' control SFR. By setting this bit, the peripheral will cease functioning when in IDLE mode. Otherwise the peripheral remains on.

17.3 Sample Code

The following code demonstrates sleep mode and the use of the watchdog timer. The WDT acts as both a guard against infinite loops and as a way to wake the PIC32 from sleep.

Code Sample 17.1. `wdt.c` Watchdog timer demonstrations.

```
#include "NU32.h"          // config bits, constants, funcs for startup and UART

int main(void) {
  if(RCONbits.WDTO) { // did we reset b/c of the watchdog timer
      if(RCONbits.SLEEP) { // was the pic in sleep mode when we reset
          __asm__ __volatile__ ("iret"); // if so, resume where we left off
```

```
        }
  }

  {
    char letters[2] = "a";
    int pressed = 0; //true if button pressed during the delay
    NU32_Startup();
    OSCCONbits.SLPEN = 1; // set power saving mode to sleep
    WDTCONbits.ON = 1;    // turn on the watchdog timer
    // Check if reset was due to WDT in idle mode
    // if so, resume operation
    // otherwise, the wdt timed out while not idle, so start from the beginning
    // enable the watch dog timer
    // watchdog timer resets after 4.096 seconds  (this setting is a config bit set by bootloader,
    // it cannot be changed in software)
    while(1) {
      if(!NU32_USER || pressed) { // user button pushed (it's low if pressed)
        if(letters[0] < 'j') {
          __asm__ __volatile__ ("wait"); // we are early, so sleep.  Watchdog will wake us up
          //and the program will resume where it left off
        } else {
          //we got stuck in an infinite loop
          while(1) {
            _nop();// fortunately the watchdog timer will reset the pic
          }
        }
      }
      NU32_WriteUART1(letters);
      ++letters[0];
      pressed = 0;
      _CP0_SET_COUNT(0);
      while(_CP0_GET_COUNT() < 80000000) {
        pressed |= !NU32_USER; // delay for ~1 second, still poll the user button
      }
      // clear the watchdog timer
      WDTCONbits.WDTCLR = 1;
    }
  }
  return 0;
}
```

First, the code enables the WDT and sleep mode. Next, the PIC32 checks for what caused its last reset. If the WDT caused the PIC32 to reset while in sleep mode, an `iret` instruction is issued, allowing the code to resume from where it left off. Otherwise, the WDT reset came due to an infinite loop, so the program starts from the beginning.

Next, the program enters a loop, printing the alphabet to the serial terminal at approximately one letter per second. After each letter, the software writes to WDTCONbits.WDTCLR, resetting the watchdog timer.

Pressing the USER button prior to the letter "j" being printed causes the PIC32 to enter an infinite loop that does nothing. As the code no longer sets WDTCONbits.WDTCLR, the WDT will eventually time out, resetting the PIC32. Upon resetting, the program will start printing the letters from the beginning. When the PIC32 resets in this manner it typically indicates a bug exists that should be fixed. Pressing the USER button after the letter "j" causes the PIC32 to enter sleep mode. While the PIC32 sleeps the WDT continues to tick but WDTCONbits.WDTCLR is no longer set; thus the WDT will timeout and reset the PIC32. The code detects that the PIC32 has awoken, and, rather than restarting the program, resumes from where it left off.

17.4 Chapter Summary

- Lower oscillator frequencies result in lower power consumption but also reduced performance.
- In idle mode, the CPU stops executing instructions but most peripherals can continue to operate. Interrupts or the WDT can wake the PIC32 from idle.
- In sleep mode, the CPU stops executing instructions and only a few peripherals can operate . External interrupts or the WDT can wake the PIC32 from sleep.
- The watchdog timer can reset the PIC32 when it gets stuck due to a software issue, or wake the PIC32 from sleep during normal operation.
- You must periodically write to WDTCONbits.WDTCLR to reset the WDT; otherwise the PIC32 will reset.
- Important watchdog timer settings must be set in the configuration bits, which prevents software from modifying them.
- Software can check RCON at reset to determine whether the program has started due to the WDT or for other reasons. A reset due to the WDT usually indicates a bug in the code, and is implemented as a safety measure to prevent the PIC32 from permanently freezing due to a software mistake.

Chapter 18

Flash Memory

Flash memory retains its contents, even when powered off. So far, we have used this non-volatile memory (NVM) to store your programs and the bootloader; however, we have not explicitly accessed it from software. By writing to the appropriate SFRs, software can write data to flash memory, allowing you to save settings that persist even when the PIC32 loses power. The process of writing to the flash via software, outlined here, is often referred to as run-time self programming (RTSP). The bootloader uses RTSP to store your programs on the PIC32.

Flash can also be written using an external programmer such as the PICkit 3. This chapter does not discuss how external programmers work, however, their operation is described in the PIC32 Flash Programming Specification. The bootloader was originally stored on your PIC32 using an external programmer.

18.1 Overview

The PIC32 contains 512kB of flash program memory and 12 kB of boot flash. The program memory is divided into the hierarchical structure depicted in Fig. 18.1. The largest unit of flash memory is one page, the equivalent of 4096 bytes. Pages are divided into eight rows, each containing 512 bytes. The rows contain 128 four byte words, the smallest unit of flash memory.

As per the PIC32 memory model (see Ch. 2), each byte of flash resides at a unique physical address. The CPU (or the prefetch cache module) can directly read from flash memory during the execution of your program. If you have ever declared any `const` global variables, these were stored in flash and accessed by the CPU without needing any special instructions.[1]

Unlike reading from flash, writing to flash requires an explicit multi-step effort by the programmer. Two operations are required to write data to flash: erase and write. Erase sets all bits to one and can only be applied to a single page or all pages. Writes can switch individual bits from one to zero but not the other way around. Either a whole row or a single word can be written in a single operation.

The ability to write to flash from software allows you store persistent values on the PIC32; however, special precautions must be used when writing to the flash. Flash memory fails after too many erase-write cycles; therefore, you should minimize the number of writes. Additionally, program code resides in flash, so incorrect writes can change the code itself! To prevent you from accidentally writing to the flash, you must perform an unlock sequence prior to any write. Given these limitations, only write to flash memory that does not contain program instructions and write as infrequently as possible.

18.2 Details

The mechanics of writing to flash memory are controlled via the flash SFRs and described in more detail in the Flash Programming section of the PIC32 reference manual.

NVMCON The main control register for the flash memory.

[1] This behavior is specific to the XC32 compiler and is not a rule for general C programs

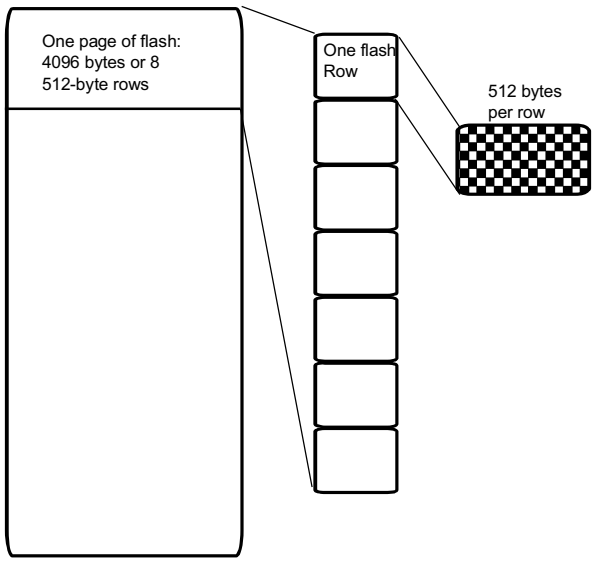


Figure 18.1: Division of flash memory into 4096 byte pages, 512 byte rows, and 4 byte words.

NVMCON⟨15⟩ or NVMCONbits.WR: Set to one to start a flash operation. When the operation completes, the PIC32 clears this bit. This bit can only be set if the flash unlock sequence has been performed and NVMCONbits.WREN = 1.

NVMCON⟨14⟩ or NVMCONbits.WREN: Write enable bit. When set, NVMCONbits.WR can be written, as long as the unlock sequence has been performed. You should set this bit prior to writing to flash and clear it when you are finished.

NVMOP⟨3:0⟩ or NVMCONbits.NVMOP: Determines the operation that is performed when NVMCONbits.WR is set.

0b0101 Erase all pages (mostly useful for bootloaders).

0b0100 Erase a single page, selected by NVMADDR.

0b0011 Write a row to the row chosen by NVMADDR. Data to be written is stored at the address in NVMSRCADDR.

0b0001 Write a word to the address stored in NVMADDR. The word to write is stored in NVMDATA.

NVMKEY Used to unlock the NVMCON register to enable erasing and writing. Issue the following commands, with interrupts disabled, to unlock NVMCON:

```
NVMKEY = 0xAA996655;
NVMKEY = 0x556699AA;
```

Typically, after performing this sequence you should immediately set NVMCONbits.WR to perform the desired flash operation:

```
NVMCONSET = 0x8000
```

If you wait to set NVMCONbits.WR your operation may fail because the unlock sequence remains valid for only one peripheral bus transaction.

NVMADDR Stores the physical address of the flash memory that the next operation will modify.

NVMDATA Stores data written to flash when a single word is written.

NVMSRCADDR The word-aligned physical address of data to be written when a whole row is programmed. Word aligned means that the physical memory address must be evenly divisible by 4 (since there are 4 bytes in a flash word).

18.3 Sample Code

The following code allocates one page of flash memory and provides functions for writing to, reading from, and erasing the flash.

Code Sample 18.1. `flash.h` Flash memory header file.

```
#ifndef FLASH__H__
#define FLASH__H__
// the flash module allocates a pgae of flash to be used as a buffer and provides read/write accesss

#define PAGE_SIZE 4096                                          // size of a page, in bytes
#define PAGE_WORDS (PAGE_SIZE/4)                                // size of a page, in words

void flash_erase(void);                         // erases the flash buffer. resets the memory to ones

void flash_write_word(unsigned int index, unsigned int data);   // writes a word (32 bits) to flash

unsigned int flash_read_word(unsigned int index);               // reads a word from flash

#endif
```

Code Sample 18.2. `flash.c` Flash memory implementation.

```
#include "flash.h"
#include <xc.h>
#include <sys/kmem.h> // contains macros for converting between physical and virtual addresses

#define OP_ERASE_PAGE 4 // erase page operation
#define OP_WRITE_WORD 1 // perform a word operation to write a single word

// by making the buffer const, it will be stored in flash memory.
// Only one page can be erased at a time, therefore, this must be a multiple of PAGE_SIZE bytes long
// the aligned attribute ensures that the page falls at an address divisible by 4096.
// initializing the array to {0} explicitly is necessary for everything to work
static const unsigned int buffer[PAGE_WORDS] __attribute__ ((__aligned__(PAGE_SIZE))) = {0};


static void flash_op(unsigned char op) { // perform the given flash operation (op is NVMOP)
  int ie = __builtin_disable_interrupts();
  NVMCONbits.NVMOP = op; // store the operation
  NVMCONbits.WREN = 1;   // enable writes to the WR bit

  // unlock sequence
  NVMKEY = 0xAA996655;
  NVMKEY = 0x556699AA;
  NVMCONSET = 0x8000; // set the WR bit to begin the operation.

  while (NVMCONbits.WR) { // wait for the operation to finish
    ;
  }
  NVMCONbits.WREN = 0;    // disable writes to the WR bit

  if (ie & 0x1) {         // re-enable interrupts if they had been disabled
    __builtin_enable_interrupts();
  }
```

```
}

void flash_erase() {                            // erase the flash buffer.   resets the memory to ones
  NVMADDR = KVA_TO_PA(buffer);                  // use the physical address of the buffer.
  flash_op(OP_ERASE_PAGE);                      // set bits 3 and 14 of NVMCON (WREN and NVMOP)
}

void flash_write_word(unsigned int index, unsigned int data) { // writes a single word to flash.
                                                               // a flash word is 32bits long. note that
                                                               // only zeros can be written.  so the sequence
                                                               // is to erase to get all ones
                                                               // then write zeros in certain places
  NVMADDR = KVA_TO_PA(buffer + index);                         // address to write
  NVMDATA = data;
  flash_op(OP_WRITE_WORD);
}

unsigned int flash_read_word(unsigned int index) {
  return buffer[index];
}
```

Examine `flash.c`. Notice the buffer declaration:

```
static const unsigned int buffer[PAGE_WORDS] __attribute__ ((__aligned__(PAGE_SIZE))) = {0x00};
```

This declaration looks similar to that of a normal global array, except for a few extra keywords. As per the XC32 documentation, a `const` global variable gets allocated in flash memory rather than RAM. The `__attribute__ ((__aligned__(PAGE_SIZE)))` code ensures that the linker places `buffer` on a page boundary; that is the address of buffer must be the start of a page in flash memory. In other words, `buffer` (equivalently `&buffer[0]`) must be evenly divisible by the page size (4096)). The `{0x00}` ensures that the array is initialized to all zeros.[2] Although declaring a `const` global variable is the easiest method, you can also allocate flash memory directly in the linker script.

The function `flash_op` handles unlocking the flash and executing the appropriate operation. The only operations we have implemented are writing a single word in the buffer and erasing the buffer.

The following code uses `flash` to demonstrate basic flash operations. After prompting the user to press ENTER, the code displays the contents of `buffer[2]`. Next it writes data to `buffer[2]`. Finally, after pressing ENTER again, the data written will be displayed. The first time you run the code, the initial data will be zero. After running the code once, 0xf0ldab1e will be written to the flash memory. If you power cycle the PIC32, you will see that the data has been retained.

Code Sample 18.3. `flash_basic.c` Basic flash memory demonstration

```
#include "NU32.h"
#include "flash.h"
// read a word from flash, write a word to flash.  repeat.

int main() {
  char msg[100];
  NU32_Startup();

  NU32_WriteUART1("Press ENTER to continue.\r\n");
  while(1) {
    NU32_ReadUART1(msg, 100);       // wait for the user to press enter
    sprintf(msg,"The flash holds 0x%x\r\n",flash_read_word(2));
    NU32_WriteUART1(msg);
```

[2] Ideally the array would be initialized to all ones, to save one erase; however, there is no standard C syntax to accomplish a non-zero initialization without explicitly writing every element.

```
    flash_erase();
    flash_write_word(2,0xf01dab1e); // write something to flash
                                    // now notice that if you turn off
                                    // the pic32 and turn it on again
                                    // it will show f01dab1e
  }

  return 0;
}
```

Although the above code works, it is highly inefficient. Remember, flash can only endure a limited number of write/erase cycles before failing. In the case of `flash_basic.c` a whole page of flash must be erased every time you want to write 4 bytes! The topic of optimizing flash memory access to limit erasures and writes is one of active research; however, a simple scheme can improve upon the situation here.

First, the whole buffer should be erased. You then divide the flash page into a series of cells. Some cells are control cells; bits in these cells determine whether other cells are occupied. When writing data, you use the control cells to find the next empty cell and write data to the empty flash area. Next you mark the cell as occupied by writing a zero to the corresponding control cell bit. After many writes the page will be full and you must erase it again. You can also use this scheme across multiple pages to lessen the wear on any single page.

18.4 Chapter Summary

- Flash memory can be used to store settings that you want to keep, even when the power is off.
- Flash can only be set to a one by erasing an entire page; writes can only selectively flip ones to zeros.
- Flash has a limited lifespan so you should minimize writes and erasures.

Chapter 19

Controller Area Network (CAN)

The controller area network (CAN) is an asynchronous two-wire bus protocol used extensively in the automotive industry and industrial automation. It allows many devices to communicate with each other at speeds of up to 1 Mbps, over distances of several meters.

19.1 Overview

19.1.1 Physical Layer

A CAN bus consists of two wires, CANH and CANL, terminated with 120 Ω resistors at either end (see Fig. 19.1). The state of the bus is read as the difference between CANH and CANL. If CANH − CANL is less than some threshold voltage, the state is said to be *recessive*, which corresponds to logic high (1). Likewise if CANH − CANL is greater than a threshold voltage, the state is said to be *dominant*, which corresponds to logic low (0). The terms "recessive" and "dominant" refer to the priority that each state has on the bus: if any device attempts to set a dominant (logic low) state, then the bus will be in that state, even if other devices want a recessive (logic high) state. This mechanism gives some bits priority over other bits, and is used to determine which node gets to transmit when multiple nodes attempt to transmit, a process called arbitration.

The PIC32 cannot directly create the differential coding that the CAN bus requires; rather, it communicates with a transceiver that converts between the PIC32's logic signals and the bus's logic signals. One such transceiver is the Microchip MCP2562; several other manufacturers produce similar chips. Figure 19.2 provides a circuit diagram for connecting the MCP2562 between the PIC32 and the CAN bus.

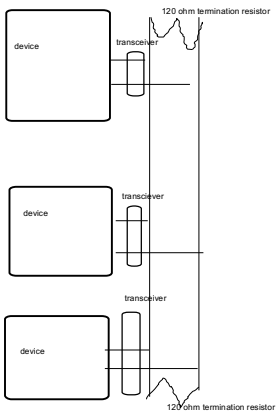


Figure 19.1: Three devices connected to the CAN bus.

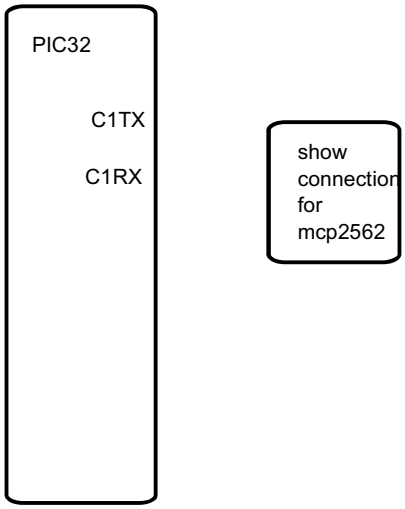
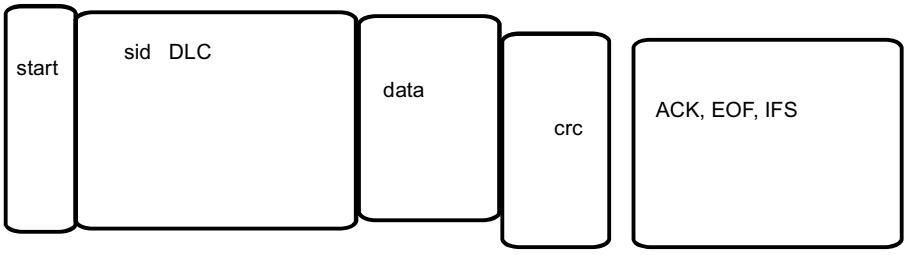


Figure 19.2: Connecting the PIC32 to the CAN bus, via the MCP2562 CAN transceiver



Standard can frame.

Figure 19.3: Layout of a standard CAN frame.

CAN employs an asynchronous protocol; like a UART, there is no explicit clock signal. Devices determine when to change and when to read a bit based on a predetermined frequency. All nodes on the network must be synchronized so that they output and interpret bits at the correct time. The CAN peripheral provides several parameters that enable devices to synchronize with each other and read bits properly. Longer wires and higher frequencies make synchronization harder to achieve due to propagation delay and oscillator inaccuracies.

19.1.2 Data-Link Layer

The CAN peripheral allows software to send and receive frames, a sequence of bits that determines the meaning and content of a message. Two types of frames exist: standard and extended. We limit the discussion to standard frames.

Figure 19.3 depicts the format for a standard CAN frame. For a programmer, the most important fields in a frame are the 11-bit identifier (SID), the data length code (DLC), and the data itself, since software must set and parse these fields. The SID determines the frame's priority on the bus: if two devices attempt to send at the same time, the frame with the lower SID will go through. Additionally, devices can choose to selectively receive or ignore frames based on their SID; in this sense it functions as an address. The DLC determines how many bytes of data can be sent. Anywhere from zero to eight bytes may be sent in a single frame.

Fields handled in hardware include the start of frame (SOF), cyclic redundancy check (CRC), acknowledge (ACK), end of frame (EOF), and inter-frame space (IFS). The SOF and EOF bits delimit the edges of a frame. The CRC is a checksum that the hardware can use to verify the integrity of the data in a frame; a checksum mismatch will cause the frame to be rejected. When a frame is sent on the CAN bus, any device that receives the data generates an acknowledgment by setting the ACK bit low. If no device acknowledges a

frame, the transmitter realizes that a transmission error has occurred and will attempt to resend the frame. Only other devices may acknowledge transmissions; therefore, the CAN bus must have at least two active devices. The IFS is the state of the bus when no device is transmitting.

Two additional fields are the remote transmit request (RTR) bit and the identifier extension (IDE) bit. The RTR bit is used to request data from another node on the network; however, its use is discouraged and we do not discuss it further [**?**]. The IDE bit, when set, indicates an extended frame.

The PIC32 handles CAN frames by using first-in first-out queues (FIFOs) stored in RAM. The CAN peripheral can use up to 32 separate FIFOs. Each FIFO can either receive (RX) or transmit (TX) CAN frames. To transmit a frame, you load data into a TX FIFO and request a transmission. The CAN module automatically places received frames into the appropriate FIFO based on the filtering process described below.

The CAN peripheral has 32 filters, all of which can be configured to trigger on a given SID. Additionally, the CAN module has four mask registers, which a filter uses to selectively ignore certain SID bits. When the CAN peripheral sees a frame on the bus, it attempts to match the SID with one of its filters. If a filter matches, CAN stores the frame in an associated RX FIFO. Thus, filters provide a hardware-based method for sorting frames into different FIFOs based on the SID. Interrupts can be triggered based on the status of each FIFO or of the CAN module as a whole.

19.2 Details

The CAN peripheral relies on several SFRs. Some apply to the CAN peripheral as a whole, while others only apply to individual filters or FIFOs. We provide the information you need to establish basic communication; the Controller Area Network Reference manual provides a full description. Unlike other peripherals, you must turn on CAN and enter a configuration mode to modify most SFRs. After setting the configuration, you change modes so that CAN can send and receive messages.

CiCON The main configuration register. Allows you to set the operating mode.

CiCON⟨26:24⟩ or CiCONbits.REQOP: Request a change in the operating mode. After requesting a mode change you must wait for the mode to actually change by polling CiCONbits.OPMOD. Switching between certain modes may require a condition on the bus to be met; thus, errors in the wiring of the bus may prevent some mode switches. The main modes are:

- 0b100 Configuration mode, which allows all CAN SFRs to be modified.
- 0b010 Loopback mode. The CAN is internally wired to itself, allowing you to test sending and receiving messages. When in loopback mode, the CAN is not actually connected to the bus; hence no transceiver is required.
- 0b000 Normal mode, the CAN peripheral operates normally and produces ACKS.

CiCON⟨23:21⟩ or CiCONbits.OPMOD: The current operating mode. Uses the same values as CiCONbits.REQOP. You should poll this value after requesting an operating mode transition to wait for CAN bus to actually enter the desired mode. The CAN peripheral cannot change modes until certain bus conditions are met; therefore, if your code hangs when polling OPMOD you may have a wiring error.

CiCON⟨15⟩ or CiCONbits.ON: Set to 1 to turn the CAN module on. You cannot switch modes unless this bit is set so, unlike other peripherals, you can enable the CAN module prior to configuring it.

CiCFG Configures the bit-rate for the CAN bus. The CAN uses time quanta T_q as its basic unit of time. A single bit period, $T_b = 1/F_{baud}$, may be anywhere from 8 to 25 T_q long, and is divided into four segments: synchronization, propagation, phase 1, and phase 2 (see Fig. 19.4). The receiver registers the value of a bit between phase 1 and phase 2. These phases allow the CAN receiver to account for oscillator differences and propagation delay. The synchronization segment is always 1 T_q, but the lengths of the other phases are configurable. The hardware may automatically adjust phase 1 or phase 2 to synchronize with other devices. When you set the baud via CiCFGbits,BRP you are actually determining the length of T_q; the total length of of all four segments determines the nominal bit frequency.

CiCFG⟨13:11⟩ or CiCFGbits.SEG1PH: The length of phase 1 is (CiCFGbits.SEG1PH + 1)T_q. In the default configuration, phase 2 will be set automatically to this value or $2T_q$, whichever is larger.

CiCFG⟨10:8⟩ or CiCFGbits.PRSEG: The length of the propagation segment is (CiCFGbits.PRSEG + 1)T_q. Should be approximately twice the length of the propagation time for a bus signal.

CiCFG⟨7:6⟩ or CiCFGbits.SJW: The maximum amount that the phase 1 or phase 2 segments can be adjusted is (CiCFGbits.SJW + 1)T_q. This value must be smaller than the length of phase 2.

CiCFG⟨5:0⟩ or CiCFGbits.BRP: Baud rate prescaler, determines the length of a time quanta. To set this value you must know the baud F_{baud} and the number of time quanta per bit N (the sum of the number of time quanta for each segment). The length of a time quanta then becomes $T_q = \frac{1}{NF_{baud}}$ and CiCFGbits.BRP = $\frac{F_{sys}T_q}{2} - 1$, where F_{sys} is the system clock frequency (80 MHz).

Setting the bit timing for CAN becomes critical when attempting to communicate quickly or over long distances. Microchip's Application Note AN754 [**?**] provides more details about choosing the proper settings for your purposes.

CiTREC Tracks the number of receive and transmit errors. This register is especially useful when debugging or in situations where managing failures is critical.

CiFLTCONr There are 7 filter control registers (r = 1 to 7). A filter directs data to various RX FIFOs based on the frame's SID. Each register contains configuration bits for several of the 32 available filters. The fields for each filter are

CiFLTCONrbits.FLTENx: Set to enable filter x.

CiFLTCONrbits.MSELx: Select the mask register for filter x. Mask registers determine which SID bits to ignore when hardware attempts to match a filter.

CiFLTCONrbits.FSElx: This value associates FIFO (0-31) with filter x. When the filter matches, the data will be stored in the specified FIFO.

CiRXFn One register per filter. Determines what the filter matches. Only operates if enabled by setting the appropriate bits in CiFLTCONr. When using standard messages, only one field is relevant:

CiRXFn⟨31:21⟩ or CiRXFnbits.SID: The SID to match.

CiRXMr There are four mask registers (r=1 to 4). Masks determine which bits of an SID to ignore. Each filter uses one mask register.

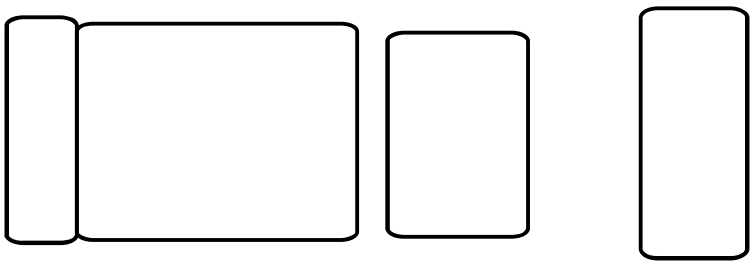
CiRXMr⟨31:21⟩ or CiRXMrbits.SID: Bits that are zero in this field will be ignored when matching a filter. Bits that are set to one will be included in filter matching. For example, if C1RXM0 = 0b101 only bits 0 and 2 matter when matching an SID against a filter.

CiFIFOBA Stores the physical address of the FIFO buffer, the location in RAM where the CAN FIFOs are located. This buffer must be large enough to store all of the FIFOs. The CAN peripheral maintains up to 31 FIFOs. Each FIFO can store between 1 and 32 messages, and each message is 16 bytes long. The FIFOs are stored contiguously in memory, as per Fig. 19.5. You can use as many or as few FIFOs as you want, and the FIFO buffer only needs to be large enough to store the used FIFOs. When deciding which FIFOs to use, start with FIFO0 and continue upwards without skipping any (i.e., if you need 2 FIFOS use FIFO0 and FIFO1 not FIFO0 and FIFO2).

CiFIFOCOn The control register for FIFO n (n = 0 to 31). Determines the length of the FIFO and whether it is for transmitting or receiving.

CiFIFOCONn⟨20:16⟩ or CiFIFOCONnbits.FSIZE: The size of the FIFO determines how many 16 byte messages it can hold and is CiFIFOCONnbits.FSIZE + 1.

CiFIFOCONn⟨14⟩ or FRESET: Set this bit to reset the FIFO. Hardware clears this bit when the FIFO after the reset process finishes. Due to hardware errors this field can only be set via CiFIFOCONnSET = 0x5000.



shows each phase of bit sampling. This is a single bit period. Maybe show relation to larger message.

Figure 19.4: Timing for sampling a single bit on the CAN bus.

CiFIFOCONn⟨13⟩ or UINC: For TX FIFOs set this bit after adding an element to increment the FIFO head. For RX FIFOs set this bit after reading an element, to increment the FIFO tail. Due to hardware errors, this field can only set via CiFIFOCONnSET = 0x2000.

CiFIFOCONn⟨7⟩ or CiFIFOCONnbits.TXEN: When set (1), the FIFO is a TX FIFO. When clear (0), the FIFO is an RX FIFO.

CiFIFOCONn⟨4⟩ or CiFIFOCONnbits.TXERR: Set when an error occurred during transmission. Cleared when read. Useful for debugging.

CiFIFOCONn⟨3⟩ or CiFIFOCONnbits.TXREQ: For TX queues, requests that the data in the queue be sent. Cleared after the message is successfully sent. The CAN peripheral repeatedly tries to send failed messages until sending is successful.

CiFIFOINTn Contains the interrupt flags (IF) and interrupt enable (IE) bits. Interrupt flags indicate the state of the FIFO and are triggered even if the particular interrupt is disabled. Thus, you need not use interrupts for these flags to be useful. Some important status flags are

CiFIFOINTn⟨10⟩ or CiFIFOINTnbits.TXNFULLIF: Read-Only. Set to one when a TX FIFO is not full, cleared to zero when a TX FIFO is full.

CiFIFOINTn⟨0⟩ or CiFIFOINTnbits.RXNEMPTYIF: Read-Only. Set to one when an RX FIFO is not empty and therefore has at least one message, clear to zero when an RX FIFO is empty.

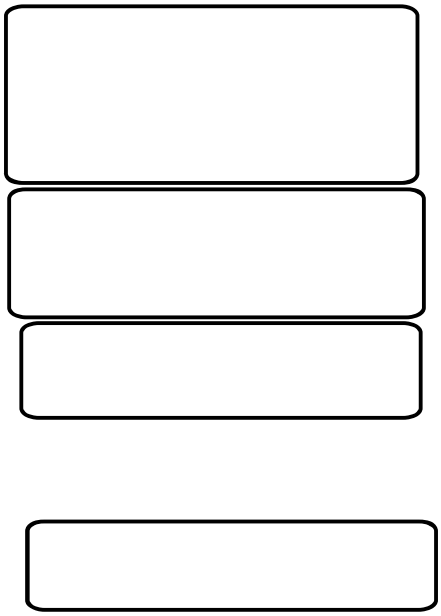
CiFIFOUAn Stores the physical address of the current position in the FIFO. For TX FIFOs, this address is where the next message will be written. For RX FIFOs, this address determines where the next message will be read.

19.2.1 Addresses

The CAN bus manages all of its FIFOs in RAM, using physical addresses. Your program, however, uses virtual addresses (see Ch. 3 for more details about the memory map). The header file `<sys/kmem.h>` provides macros for converting between physical and virtual addresses. To convert virtual addresses to physical addresses use `KVA_TO_PA(physical_address)`. To convert physical addresses to virtual addresses, such as when you set CiFIFOBA use `PA_TO_KVA1(virtual_address)`.

19.2.2 Transmitting a Message

To transmit a message you must load it onto a TX FIFO at the current address held by CiFIFOUAn. A TX message consists of four four-byte words, which can be viewed as an array of unsigned integers. We will not explore the bit-fields in detail here, but rather demonstrate how to construct a message for a simple transmission. We assume that the first address in the queue is stored in the pointer `unsigned int * addr = PA_TO_KVA1(CiFIFOUAn)`. The fields stored in the message buffer correspond to their equivalents in a CAN data frame.



Shows how the fifos are layed out in memory.

Figure 19.5: Memory layout for CAN FIFOs.

Figure 19.6 shows the full layout of a TX message; any fields we do not discuss can be cleared to zero for our purposes. The first word, called CMSGID and stored at `addr[0]`, contains the SID of the message. The SID is 11 bits long, so it can be any number from 0 to $2^{11} - 1$. Assign the SID to `addr[0]`; for example, `addr[0] = 10` sets an SID of 10. The SID is used to determine the priority (lower numbers get higher priority on the bus if multiple devices attempt to send simultaneously). Other devices on the CAN network can filter messages based on the SID.

The next word, called CMSGEID and stored at `addr[1]` contains the length (DLC) of the data in its lower 4 bits (bits 0-3). The length can be anywhere from 0 to eight bytes. Set the length of the data payload by setting `addr[1] = length`. The other fields in CMSGEID have meaning in advanced contexts, but for the simple cases here they can all be zero, which happens automatically when you assign the length to `addr[1]`.

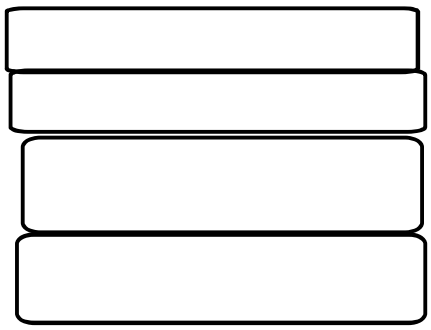
The next two words, `addr[2] (CMSGDATA0)` and `addr[3]` (CMSGDATA1), contain the data that is sent with the message. Only the DLC number of bytes will be sent. If you are sending either one or two integers you can simply assign to these arrays directly; otherwise, you will need to specifically address the bytes using either bit manipulation operations or typecasting.

Below, we describe a typical process for transmitting a CAN frame.

1. Ensure that the TX FIFO is not full by checking that CiFIFOINTnbits.TXNFULLIF = 1.
2. Get the address of the next TX message by reading CiFIFOUAn and converting it to a virtual address using `PA_TO_KVA1`.
3. Store the desired message on the FIFO.
4. Set the FIFO's UINC bit, using `CiFIFOCONnSET = 0x2000` to notify the CAN peripheral that a message has been added to the queue.
5. Set CiFIFOCONnbits.TXREQ to one, requesting transmission. This bit will be cleared by hardware when the transmission completes successfully.

19.2.3 Receiving a Message

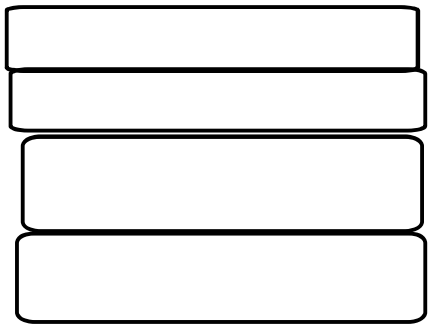
To receive a message, you must configure an RX FIFO and a filter that matches the desired message. Configuring a filter requires enabling and assigning it a filter and mask via CiFLTCONr, and setting the



shows what you need to put on the tx queue. Highlights fields that we actually set to nonzero values

Figure 19.6: Layout used by the CAN peripheral for generating a CAN frame

.



shows the fields we receive from the RX fifo

Figure 19.7: Layout used by the CAN peripheral for storing a receive CAN frame.

SID in CiRXFn. When the CAN peripheral matches the SID on the bus to an SID in a filter, it stores the message in the desired FIFO. For example, if a filter is configured with SID = 0x11 and a mask of 0x7FF, only messages with SID = 0x11 will be stored in the corresponding RX FIFO (since the mask is all ones it indicates that all bits should match exactly).

Like a transmit message, a receive message is also four four-byte words long.[1]. As before, we assume that the current RX FIFO position is stored in `addr`. Figure 19.7 shows the full layout of an RX message. The first word, called CMSGSID and stored at `addr[0]` contains the SID that was matched in the lower 11 bits. The lower four bits of the second word, called CMSGEID and stored at `addr[1]`, contains DLC, the number of bytes of data sent. The final two words, CMSGDATA0 (`addr[2]`) and CMSGDATA1 (`addr[3]`) contain the data bytes that were sent; only the first DLC bytes are valid.

To read a message you should either enable message receive interrupts or poll, waiting for CiFIFOINTnbits.RXNEMPTYIF to be set. To read the message, perform the following steps.

1. Ensure that RX filters have been set and enabled.
2. The RX FIFO should be non-empty, so check that CiFIFOINTnbits.RXNEMPTYIF is set.
3. Get the address of the next RX message by reading CiFIFOUAn and converting it to a virtual address using `PA_TO_KVA1`.
4. Process the message.
5. Set the FIFO's UINC bit, using `CiFIFOCONnSET = 0x2000` to notify the CAN peripheral that you are done with the message.

[1] There is also a mode that only stores the two data words.

19.3 Sample Code

19.3.1 Loopback

The first example uses the CAN's loopback mode, one RX FIFO and one TX FIFO, allowing the CAN to send data to itself. Loopback mode allows testing the CAN bus without worrying about physical layer issues such as transceivers, propagation delay, or bus impedance. The code creates a single filter to respond to only one specific SID. It prompts the user for data, sends and receives that data over CAN, and reports the result to the user.

Code Sample 19.1. `can_loop.c` Basic CAN loopback functionality

```
#include "NU32.h"
#include <sys/kmem.h> // used to convert between physical and virtual addresses
// Basic CAN example using Loopback mode, so this functions with no external hardware.
// prompts user to enter numbers to send via CAN.
// sends the numbers and receives them via loop-back, printing the results.

#define MY_SID 0x146  // the sid that this module responds to

#define FIFO_0_SIZE 4 // size of FIFO 0, in number of message buffers
#define FIFO_1_SIZE 2 // size of FIFO 1, in number of message buffers
#define MB_SIZE 4     // number of 4-byte integers in a message buffer

volatile unsigned int fifos[(FIFO_0_SIZE + FIFO_1_SIZE)* MB_SIZE]; // buffer for CAN FIFOs

int main() {
  char buffer[100];
  NU32_Startup();

  C1CONbits.ON = 1;   // turn on the can module.
  C1CONbits.REQOP = 4; // request configure mode
  while(C1CONbits.OPMOD != 4) { ; } // wait to enter config mode

                                     // setup the fifos
  C1FIFOCON0bits.FSIZE = FIFO_0_SIZE-1;// set fifo 0 size.  Actual size is 1 + FSIZE
  C1FIFOCON0bits.TXEN = 0;            // fifo 0 is an RX fifo

  C1FIFOCON1bits.FSIZE = FIFO_1_SIZE-1; // set fifo 1 size. Actual size is 1 + FSIZE
  C1FIFOCON1bits.TXEN = 1;           // fifo 1 is a TX fifo
  C1FIFOBA = KVA_TO_PA(fifos);       // tell CAN where the fifos are

  C1RXM0bits.SID = 0x7FF;            // mask 0 requires all SID bits to match

  C1FLTCON0bits.FSEL0 = 0;           // filter 0 is for FIFO 0
  C1FLTCON0bits.MSEL0 = 0;           // filter 0 uses mask 0
  C1RXF0bits.SID = MY_SID;           // filter 0 matches against SID
  C1FLTCON0bits.FLTEN0 = 1;          // enable filter 0

                                     // we can skip the baud settings because
                                     // we only want to use loopback mode.
                                     // in other modes you must set a baud
  C1CONbits.REQOP = 2;               // request loopback mode
  while(C1CONbits.OPMOD != 2) { ; }  // wait for loopback mode

  while(1) {
    int to_send = 0;
    unsigned int * addr;             // used for storing fifo addresses
```

```
    NU32_WriteUART1("Enter number to send via CAN:\r\n");
    NU32_ReadUART1(buffer,100);
    sscanf(buffer,"%d", &to_send);
    sprintf(buffer,"Sending: %d\r\n",to_send);
    NU32_WriteUART1(buffer);

    addr = PA_TO_KVA1(C1FIFOUA1);   // get FIFO 1 (the TX fifo) current message buffer address
    addr[0] = MY_SID;               // only the sid must be set for this example
    addr[1] = sizeof(to_send);      // only DLC field must be set, we indicate 4 bytes
    addr[2] = to_send;              // 4 bytes of actual data
    C1FIFOCON1SET = 0x2000;         // C1FIFOCON1bits.UINC = 1 tells fifo to increment pointer.
                                    // Due to errata, must use C1FIFOCON1SET explicitly
                                    // here (sets the 13th bit)
    C1FIFOCON1bits.TXREQ = 1;       // request that data from the queue be sent

    while(!C1FIFOINT0bits.RXNEMPTYIF) { ; } // wait to receive data
    addr = PA_TO_KVA1(C1FIFOUA0);   // get the VA of the RX FIFO, FIFO0
    sprintf(buffer,"Received %d with SID = 0x%x\r\n",addr[2], addr[0] & 0x7FF);
    NU32_WriteUART1(buffer);
    C1FIFOCON0SET = 0x2000;        //C1FIFOCON0bits.UINC = 1 tells fifo we are done with received data.
                                   //Need to use C1FIFOCON0SET due to errata
  }
  return 0;
}
```

19.3.2 Light Control

The next example requires at least two PIC32s to be connected to transceiver chips and then to the CAN bus. The example assumes that you use the Microchip MCP2562, and connected everything as per Fig. 19.2; however, other CAN transceivers will work. One PIC32 runs `can_cop.c`, which "simulates" a police officer attempting to control a traffic light. This PIC32 communicates with the computer over UART, allowing you to specify the light color (Red, Yellow, or Green).

Code Sample 19.2. `can_cop.c` Control the lights on another PIC32 via CAN.

```
#include "NU32.h"
#include <sys/kmem.h> // used to convert between physical and virtual addresses
#include <ctype.h>    // tolower
// The CAN cop is the "police officer" that controls the traffic light

#define FIFO_0_SIZE 4 // size of FIFO 0, in number of message buffers
#define FIFO_1_SIZE 4
#define MB_SIZE 4     // number of 4-byte integers in a message buffer

#define LIGHT1_SID 1  // sid of the first traffic light

volatile unsigned int fifos[(FIFO_0_SIZE + FIFO_1_SIZE)* MB_SIZE]; // buffer for CAN FIFOs

int main() {
  char buffer[100];
  NU32_Startup();

  C1CONbits.ON = 1;   // turn on the can module.
  C1CONbits.REQOP = 4; // request configure mode
  while(C1CONbits.OPMOD != 4) { ; } // wait to enter config mode

                                    // setup the fifos
```

```
  C1FIFOCON0bits.FSIZE = FIFO_0_SIZE-1;// set fifo 0 size
  C1FIFOCON0bits.TXEN = 0;             // fifo 0 is an RX fifo

  C1FIFOCON1bits.FSIZE = FIFO_1_SIZE-1;// set fifo 1 size
  C1FIFOCON1bits.TXEN = 1;             // fifo 1 is a TX fifo
  C1FIFOBA = KVA_TO_PA(fifos);         // tell CAN where the fifos are

                        //baud setup.  Many parameters depend on physical properties of the system,
                        // We are assuming very short distance communication
                        // with similar devices so the details do not matter much
  C1CFGbits.BRP = 16;   // BRP = FSYS/(2*Ftq) - 1 = 80,000,000/(2*2,500,000) - 1

                        // assign time quanta to each phase, so the total is 16.
                        // time quanta help nodes synchronize in light of propagation delay
                        // and phase/frequency errors.
                        // 1 tq(sync) + 3 tq(propagation) + 3 tq(phase 1) + 3 tq(phase 2)  = 10 tq
  C1CFGbits.PRSEG =  3; // number of Tq in preadjust phase
  C1CFGbits.SEG1PH = 3; // # of Tq in phase 1. Phase 2 is set automatically (to 3)
  C1CFGbits.SJW = 0;    // number of Tq is SJW +1 adjustments allowed when re-syncing
                        // after being configured, you now need to set the appropriate mode.

  C1CONbits.REQOP = 0;                // request normal mode
  while(C1CONbits.OPMOD != 0) { ; }   // wait for normal mode
  NU32_LED1 = 0;
  while(1) {
    char cmd;
    unsigned int * addr;              // used for storing fifo addresses
    NU32_WriteUART1("(R)ed, (Y)ellow, or (G)reen?\r\n");
    NU32_ReadUART1(buffer,100);
    sscanf(buffer,"%c", &cmd);
    sprintf(buffer,"Setting %c\r\n", cmd);
    NU32_WriteUART1(buffer);

    if(C1TRECbits.TXWARN) {                             // many bad transmissions have occurred,
      sprintf(buffer,"Error: C1TREC 0x%08x\r\n",C1TREC); // print info to help debug the physical connection
      NU32_WriteUART1(buffer);                          // If the TX error count is high
                                                        // you may not be getting ACKs.
    }

    addr = PA_TO_KVA1(C1FIFOUA1); // get FIFO 1 (the TX fifo) current message buffer address
    addr[0] = LIGHT1_SID;         // only the sid must be set for this example
    addr[1] = sizeof(cmd);        // only DLC field must be set, we indicate the size of the command (1 byte)
    addr[2] = tolower(cmd);       // the actual data
    C1FIFOCON1SET = 0x2000;       // C1FIFOCON1bits.UINC = 1 tells fifo to increment pointer.
                                  // Due to errata, must use C1FIFOCON1SET explicitly here (sets the 13th bit)
    C1FIFOCON1bits.TXREQ = 1;     // request that data from the queue be sent

  }
  return 0;
}
```

Note that `can_cop.c` performs some rudimentary error checking and prints some diagnostic information. If you see such information, you should confirm that you have wired the bus correctly. If you run `can_cop.c` without any other devices on the CAN bus, you will eventually trigger the error condition, as CAN requires at least two devices on the bus for acknowledgment generation.

The `can_cop` can control multiple CAN traffic lights. These PIC32s change their lights to red (both

LEDs off), yellow (one LED on, one LED off), or green (both LEDs on), in response to commands from the `can_cop`.

Code Sample 19.3. `can_light.c` The `can_cop.c` program can control the LED status via CAN.

```
#include "NU32.h"
#include <sys/kmem.h> // used to convert between physical and virtual addresses
// simulates a traffic light that can be controlled via can_cop
// LED1 and LED2 on = GREEN
// LED1 on LED2 off = YELLOW
// LED1 and LED2 off = RED

#define FIFO_0_SIZE 4 // size of FIFO 0, in number of message buffers
#define FIFO_1_SIZE 4
#define MB_SIZE 4     // number of 4-byte integers in a message buffer

#define LIGHT1_SID 1  // sid of the first traffic light

volatile unsigned int fifos[(FIFO_0_SIZE + FIFO_1_SIZE)* MB_SIZE]; // buffer for CAN FIFOs

int main() {
  NU32_Startup();

  C1CONbits.ON = 1;   // turn on the can module.
  C1CONbits.REQOP = 4; // request configure mode
  while(C1CONbits.OPMOD != 4) { ; } // wait to enter config mode

                                    // setup the fifos
  C1FIFOCON0bits.FSIZE = FIFO_0_SIZE-1; // set fifo 0 size
  C1FIFOCON0bits.TXEN = 0;              // fifo 0 is an RX fifo

  C1FIFOCON1bits.FSIZE = FIFO_1_SIZE-1; // set fifo 1 size
  C1FIFOCON1bits.TXEN = 1;              // fifo 1 is a TX fifo
  C1FIFOBA = KVA_TO_PA(fifos);          // tell CAN where the fifos are


  C1RXM0bits.SID = 0x7FF;       // mask 0 requires all SID bits to match

  C1FLTCON0bits.FSEL0 = 0;      // filter 0 is for FIFO 0
  C1FLTCON0bits.MSEL0 = 0;      // filter 0 uses mask 0
  C1RXF0bits.SID = LIGHT1_SID;  // filter 0 matches against SID
  C1FLTCON0bits.FLTEN0 = 1;     // enable filter 0

                                // baud setup.  Many parameters depend on physical properties of the system,
                                // We are assuming very short distance communication with similar
                                // devices so the details do not matter much
  C1CFGbits.BRP = 16;           // BRP = FSYS/(2*Ftq) - 1 = 80,000,000/(2*2,500,000) - 1

                                // assign time quanta to each phase, so the total is 16.
                                // time quanta help nodes synchronize in light of propagation delay
                                // and phase/frequency errors.
                                // 1 tq(sync) + 3 tq(propagation) + 3 tq(phase 1) + 3 tq(phase 2)  = 10 tq
  C1CFGbits.PRSEG =  3;         // number of Tq in preadjust phase
  C1CFGbits.SEG1PH = 3;         // # of Tq in phase 1. Phase 2 is set automatically (to 3)
  C1CFGbits.SJW = 0;            // number of Tq is SJW +1 adjustments allowed when re-syncing
                                // after being configured, you now need to set the appropriate mode.

  C1CONbits.REQOP = 0;               // request normal mode
```

```
  while(C1CONbits.OPMOD != 0) { ; }   // wait for normal mode
  NU32_LED1 = 1;
  NU32_LED2 = 1;
  while(1) {
    unsigned int * addr;
    if(C1FIFOINT0bits.RXNEMPTYIF) {   // we have received data
      addr = PA_TO_KVA1(C1FIFOUA0);   // get the VA of the RX fifo, fif0 0
      switch(addr[2]) {
        case 'r':                     // switch to red
          NU32_LED1 = 1;
          NU32_LED2 = 1;
          break;
        case 'y':
          NU32_LED1 = 0;
          NU32_LED2 = 1;
          break;
        case 'g':
          NU32_LED1 = 0;
          NU32_LED2 = 0;
          break;
        default:
          ;// error! In real life, do something here
      }
      C1FIFOCON0SET = 0x2000;   // C1FIFOCON0bits.UINC = 1 tells fifo
                                // we are done with received data.
                                // Need to use C1FIFOCON0SET due to errata
    }
  }
  return 0;
}
```

19.4 Chapter Summary

- CAN is an asynchronous protocol, developed for and used extensively in the automotive industry. It is also used in other industrial control systems.
- The CAN peripheral has several operating modes. You must request a mode and then wait for it to enter the appropriate mode before continuing.
- The CAN peripheral operates on FIFOs stored in RAM. A FIFO can be either an RX or TX FIFO.
- All CAN devices on the same bus receive and acknowledge all messages. Hardware filters, based on the SID, can determine whether data gets stored into an RX FIFO. Different SIDs can be stored in different FIFOs.
- Choosing the bit timing for the CAN bus is an involved process and requires you to know about the physical properties of the bus. Proper bit timing is crucial for CAN to operate at long distances and high speeds.