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 systems due to their relative simplicity. A UART can be used with an external transceiver device to implement RS-232 communication, RS-485 multipoint communication, IrDA infrared wireless communication, or other types of wireless communication such as the IEEE 802.15.4 standard.

11.1 Overview

The PIC32 has six UARTs, each allowing it to communicate with one other device. Each UART uses at least two pins, one for receiving data (RX) and one for transmitting data (TX). Additionally, the devices share a common ground (GND) line. A UART can simultaneously send and receive data, a feature known as full duplex communication. For one-way communication, only one wire (in addition to GND) is required. To distinguish your PIC32 from the device with which it is communicating, which may be your computer or another PIC32, we call the other device *data terminal equipment* (DTE). The RX line for the PIC32 is the TX line for the DTE, and the TX line for the PIC32 is the RX line for the DTE.

You have used the PIC32's UARTs to communicate with your computer. FTDI driver software on your computer sends data over a USB cable, where a chip on the NU32 (the FTDI FT231X) receives the USB data and converts it into signals appropriate for one of the PIC32's UARTs. Data sent by the PIC32's UART is converted by the FTDI chip to USB signals to send to your computer, where the FTDI driver software interprets it as if received by a UART on your computer.

The important parameters for UART communication are the baud, the data length, the parity, and the number of stop bits. The two devices use the same parameters for successful communication. The baud refers to the number of bits sent per second. The PIC32's UART sends and receives data in groups of 8 or 9 bits. For data lengths of 8 bits, the PIC32 can

Embedded Computing and Mechatronics with the PIC32 Microcontroller. http://dx.doi.org/10.1016/B978-0-12-420165-1.00011-1

optionally transmit an additional parity bit as a simple transmission error-detection measure. For example, if the parity is "even," the number of bits sent that are one must be an even number; the parity bit is chosen to meet this constraint. If the receiver sees an odd number of ones in the transmission, then it knows a transmission error has occurred. Finally, the PIC32's UART can be set to one or two stop bits, which are ones sent at the end of a transmission.

The NU32 library uses a baud of 230,400, eight data bits, no parity bit, and one stop bit. Written in shorthand, this is 230,400/8N1. Parity may be odd, even, or none.

For historical reasons, common baud choices include 1200, 2400, 4800, 9600, 19,200, 38,400, 57,600, 115,200, and 230,400, but any choice is possible, as long as both devices agree. According to the Reference Manual, the PIC32's UART is theoretically capable of baud up to 20 M; however, in practice, the maximum achievable baud is much lower.

UART communication is asynchronous, meaning that there is no clock line to keep the two devices in sync. Due to differences in clock frequencies on the two devices, the baud for each device may be slightly different, and UART devices can handle slight differences by resynchronizing their baud clocks on each transmission.

Figure 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 least-significant bit first (e.g., the first bit sent for 0b11001000 will be a zero). Following the data bits, a parity bit may be optionally sent. Finally, the transmitter holds the line high, transmitting one or two 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. The start 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

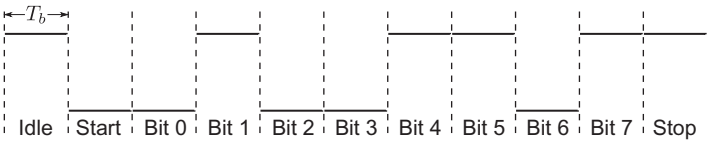

Figure 11.1
UART transmission of 0b10110010 with 8 data bits, no parity, and one stop bit.

another byte needs to be added, an overrun condition occurs and the data is lost. To prevent a TX FIFO overrun, software should 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 data arrives. Hardware maintains flags indicating the status of the FIFOs and can also interrupt based on the number of items in the FIFOs.

An optional 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 ($\overline{\text{CTS}}$).[1] When the RX FIFO is full, the UART hardware de-asserts (drives high) $\overline{\text{RTS}}$, which tells the DTE not to send data. When the RX FIFO has space available, the hardware asserts (drives low) $\overline{\text{RTS}}$, allowing the DTE to send data. The DTE controls $\overline{\text{CTS}}$. When the DTE de-asserts (drives high) $\overline{\text{CTS}}$, 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.

These are 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 a specific device, read its data sheet and select the appropriate options.

11.2 Details

Below is a description of the UART registers. The "x" in the SFR names stands for UART number 1 to 6. All bits default to zero except for two read-only bits in UxSTA.

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: This is called the "high baud rate generator bit" and controls the value of a divisor M used in calculating the baud rate (see the SFR UxBRG). If this bit is 1, $M = 4$, and if it is 0, $M = 16$.

[1] Some UARTs on the PIC32 do not have hardware flow control lines, and the flow control pins of one UART may coincide with the RX and TX lines of another UART. For example, using UART3 with flow control prevents the use of UART6.

UxMODE⟨2:1⟩ or UxMODEbits.PDSEL: Determines the parity and number of data bits.
- 0b11 9 data bits, no parity.
- 0b10 8 data bits, odd parity.
- 0b01 8 data bits, even parity.
- 0b00 8 data bits, 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 eight bytes in its TX FIFO. Interrupts will continue to happen until the condition causing the interrupt ends.
- 0b10 Interrupt while TX FIFO is empty.
- 0b01 Interrupt after everything in the TX FIFO has been transmitted.
- 0b00 Interrupt whenever the TX 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 eight bytes in its RX FIFO. The interrupt will continue to happen until the condition causing the interrupt is cleared.
- 0b10 Interrupt whenever the RX FIFO contains six or more characters.
- 0b01 Interrupt whenever the RX FIFO contains four or more characters.
- 0b00 Interrupt whenever the RX FIFO 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, 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 eight-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 Use this SFR to receive data. Hardware shifts received data bit by bit into an internal RX shift register. After receiving a full byte, hardware transfers it from the shift register into the RX FIFO. Reading from UxRXREG removes a byte from the RX FIFO. If you do not read from UxRXREG often enough, the RX FIFO may overrun. If the FIFO is full, subsequent received bytes are discarded and an overrun error status flag is set.

UxBRG Controls the baud. The value for this register should be set to achieve the desired baud B according to the following equation:

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

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

Interrupt vector numbers for the UARTs are named `_UART_x_VECTOR`, where `x` is 1 to 6. The interrupt flag status bits for UART1 are IFS0bits.U1EIF (error interrupt generated by a parity error, framing error, or overrun error), IFS0bits.U1RXIF (RX interrupt), and IFS0bits.U1TXIF (TX interrupt). The interrupt enable control bits for UART1 are IEC0bits.U1EIE (error interrupt enable), IEC0bits.U1RXIE (RX interrupt enable), and IEC0bits.U1TXIE (TX interrupt enable). The priority and subpriority bits are IPC6bits.U1IP and IPC6bits.U1IS. Interrupt flag status bits, enable control bits, and priority bits for UART2 are named similarly (replacing "U1" with "U2") and are in IFS1, IEC1, and IPC8; for UART3 they are in IFS1, IEC1, and IPC7; and for UART4 to UART6 they are in IFS2, IEC2, and IPC12.

11.3 Sample Code

11.3.1 Loopback

In our first example, the PIC32 uses UART1 to talk to itself. Connect U1RX (RD2) to U1TX (RD3). The program uses the NU32 library and UART3 to prompt the user for a single byte,

sends it twice from U1TX to U1RX, and reports the byte that was read on U1RX. We set the baud to an extremely low rate (100) so that you can easily see the transmission on an oscilloscope. If you set the oscilloscope into single capture mode and trigger on the falling edge, the scope will capture the signal from the beginning of the transmission, when the first start bit is sent. Sending the byte twice allows you to verify the stop bits.

Code Sample 11.1 uart_loop.c**. UART Code that Talks to Itself.**

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

// We will set up UART1 at a slow baud rate so you can examine the signal on a scope.
// Connect the UART1 RX and TX pins together so the UART can communicate with itself.

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

  // initialize UART1:  100 baud, odd parity, 1 stop bit
  U1MODEbits.PDSEL = 0x2; // odd parity (parity bit set to make the number of 1's odd)
  U1STAbits.UTXEN = 1;    // enable transmit
  U1STAbits.URXEN = 1;    // enable receive

  // U1BRG = Fpb/(M * baud) - 1  (note U1MODEbits.BRGH = 0 by default, so M = 16)
  // setup for 100 baud.  This means 100 bits /sec or 1 bit/ 1/10ms
  U1BRG = 49999;     // 80 M/(16*100) - 1 = 49,999
  U1MODEbits.ON = 1; // turn on the uart

  // scope instructions: 10 ms/div, trigger on falling edge, single capture
  while(1) {
    unsigned char data = 0;
    NU32_WriteUART3("Enter hex byte (lowercase) to send to UART1 (i.e., 0xa1): ");
    NU32_ReadUART3(msg, sizeof(msg));
    sscanf(msg,"%2x",&data);
    sprintf(msg,"0x%02x\r\n",data);
    NU32_WriteUART3(msg); //echo back

    while(U1STAbits.UTXBF) {  // wait for UART to be ready to transmit
      ;
    }
    U1TXREG = data;           // write twice so we can see the stop bit
    U1TXREG = data;
    while(!U1STAbits.URXDA) { // poll to see if there is data to read in RX FIFO
      ;
    }
    data = U1RXREG;           // data has arrived; read the byte
    while(!U1STAbits.URXDA) { // wait until there is more data to read in RX FIFO
      ;
    }
    data = U1RXREG;           // overwriting data from previous read! could check if same
    sprintf(msg,"Read 0x%x from UART1\r\n",data);
    NU32_WriteUART3(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 when an error has occurred. For example, you can interrupt when 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 reads data from your terminal emulator and sends it back. It uses UART3, as does the NU32 library, but does not use the NU32 UART commands. An interrupt is triggered when the RX buffer contains at least one character, and the ISR immediately sends the data back to the terminal emulator.

Using interrupts for serial I/O allows the PIC32 to receive data from the serial port without wasting time polling for it.

Code Sample 11.2 uart_int.c**. UART Code that Uses Interrupts to Receive Data.**

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

void __ISR(_UART_3_VECTOR, IPL1SOFT) IntUart1Handler(void) {
  if (IFS1bits.U3RXIF) {       // check if interrupt generated by a RX event
    U3TXREG = U3RXREG;         // send the received data out
    IFS1bits.U3RXIF = 0;       // clear the RX interrupt flag
  } else if(IFS1bits.U3TXIF) { // if it is a TX interrupt
  } else if(IFS1bits.U3EIF) {  // if it is an error interrupt. check U3STA for reason
  }
}

int main(void) {
  NU32_Startup();   // cache on, interrupts on, LED/button init, UART init
  NU32_LED1 = 1;
  NU32_LED2 = 1;
  __builtin_disable_interrupts();

  // set baud to 230400, to match terminal emulator; use default 8N1 of UART
  U3MODEbits.BRGH = 0;
  U3BRG = ((NU32_SYS_FREQ / 230400) / 16) - 1;

  // configure TX & RX pins
  U3STAbits.UTXEN = 1;
  U3STAbits.URXEN = 1;

  // configure using RTS and CTS
  U3MODEbits.UEN = 2;

  // configure the UART interrupts
  U3STAbits.URXISEL = 0x0; // RX interrupt when receive buffer not empty
  IFS1bits.U3RXIF = 0;     // clear the rx interrupt flag.  for
                           // tx or error interrupts you would also need to clear
                           // the respective flags
```

```
  IPC7bits.U3IP = 1;       // interrupt priority
  IEC1bits.U3RXIE = 1;     // enable the RX interrupt

  // turn on UART1
  U3MODEbits.ON = 1;
  __builtin_enable_interrupts();
  while(1) {
    ;
  }
  return 0;
}
```

11.3.3 NU32 Library

The NU32 library contains three functions that access the UART: `NU32_Setup`, `NU32_ReadUART3`, and `NU32_WriteUART3`. The setup code configures UART3 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_ReadUART3` 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_WriteUART3` waits for the TX FIFO to have available space before attempting to add more data to it. Also, since hardware flow control is enabled on the PIC32's UART, no data will be sent by the PIC32 unless the DTE (your computer) holds the $\overline{\text{CTS}}$ line low. The terminal emulator must have hardware flow control enabled to ensure correct operation.

Code Sample 11.3 `NU32.c`**. The NU32 Library Implementation.**

```
#include "NU32.h"

// Device Configuration Registers
// These only have an effect for standalone programs but don't harm bootloaded programs.
// the settings here are the same as those used by the bootloader
#pragma config DEBUG = OFF          // Background Debugger disabled
#pragma config FWDTEN = OFF         // WD timer: OFF
#pragma config WDTPS = PS4096       // WD period: 4.096 sec
#pragma config POSCMOD = HS         // Primary Oscillator Mode: High Speed crystal
#pragma config FNOSC = PRIPLL       // Oscillator Selection: Primary oscillator w/ PLL
#pragma config FPLLMUL = MUL_20     // PLL Multiplier: Multiply by 20
#pragma config FPLLIDIV = DIV_2     // PLL Input Divider:  Divide by 2
#pragma config FPLLODIV = DIV_1     // PLL Output Divider: Divide by 1
#pragma config FPBDIV = DIV_1       // Peripheral Bus Clock: Divide by 1
#pragma config UPLLEN = ON          // USB clock uses PLL
#pragma config UPLLIDIV = DIV_2     // Divide 8 MHz input by 2, mult by 12 for 48 MHz
#pragma config FUSBIDIO = ON        // USBID controlled by USB peripheral when it is on
#pragma config FVBUSONIO = ON       // VBUSON controlled by USB peripheral when it is on
#pragma config FSOSCEN = OFF        // Disable second osc to get pins back
#pragma config BWP = ON             // Boot flash write protect: ON
#pragma config ICESEL = ICS_PGx2    // ICE pins configured on PGx2
#pragma config FCANIO = OFF         // Use alternate CAN pins
#pragma config FMIIEN = OFF         // Use RMII (not MII) for ethernet
```

```
#pragma config FSRSSEL = PRIORITY_6 // Shadow Register Set for interrupt priority 6

#define NU32_DESIRED_BAUD 230400    // Baudrate for RS232

// Perform startup routines:
//  Make NU32_LED1 and NU32_LED2 pins outputs (NU32_USER is by default an input)
//  Initialize the serial port - UART3 (no interrupt)
//  Enable interrupts
void NU32_Startup() {
  // disable interrupts
  __builtin_disable_interrupts();

  // enable the cache
  // 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
  // to disable cache, use 0xa4210582
  __builtin_mtc0(_CP0_CONFIG, _CP0_CONFIG_SELECT, 0xa4210583);

  // 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 B10, B11, B12 and B13 back
  DDPCONbits.JTAGEN = 0;

  TRISFCLR = 0x0003;  // Make F0 and F1 outputs (LED1 and LED2)
  NU32_LED1 = 1;      // LED1 is off
  NU32_LED2 = 0;      // LED2 is on

  // turn on UART3 without an interrupt
  U3MODEbits.BRGH = 0; // set baud to NU32_DESIRED_BAUD
  U3BRG = ((NU32_SYS_FREQ / NU32_DESIRED_BAUD) / 16) - 1;

  // 8 bit, no parity bit, and 1 stop bit (8N1 setup)
  U3MODEbits.PDSEL = 0;
  U3MODEbits.STSEL = 0;

  // configure TX & RX pins as output & input pins
  U3STAbits.UTXEN = 1;
  U3STAbits.URXEN = 1;
  // configure hardware flow control using RTS and CTS
  U3MODEbits.UEN = 2;

  // enable the uart
  U3MODEbits.ON = 1;

  __builtin_enable_interrupts();
```

```
}

// Read from UART3
// 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_ReadUART3(char * message, int maxLength) {
  char data = 0;
  int complete = 0, num_bytes = 0;
  // loop until you get a '\r' or '\n'
  while (!complete) {
    if (U3STAbits.URXDA) { // if data is available
      data = U3RXREG;      // 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 character array using UART3
void NU32_WriteUART3(const char * string) {
  while (*string != '\0') {
    while (U3STAbits.UTXBF) {
      ; // wait until tx buffer isn't full
    }
    U3TXREG = *string;
    ++string;
  }
}
```

11.3.4 Sending Data from an ISR

It is often desirable to stream data collected by the PIC32 to your computer. For example, a fixed-frequency ISR could sample data from a sensor and then send it back to your computer for plotting. The ISR may collect samples at a much higher rate than they can be sent over the UART, however. In this case, some of the data has to be discarded. The process of keeping only one piece of data per every N collected (discarding $N - 1$) is called *decimation*.[2] Decimation is common in signal processing.

In addition to decimation, another concept that enables streaming data from an ISR is that of a *circular buffer*. A circular (or ring) buffer is an implementation of a FIFO, such as the UART's

[2] The origin of this term is a disciplinary practice of the Roman Army, whereby one out of every ten soldiers who had performed disgracefully was killed.

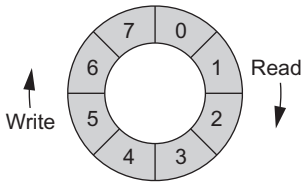

Figure 11.2
An eight-element circular buffer (FIFO), where the `write` index currently points to element 5 and the `read` index currently points to element 1.

TX and RX FIFOs. A circular buffer is implemented as an array and two index variables: `write` and `read` (see Figure 11.2). Data is added to the array at the `write` location and read from the `read` location, after which the indexes are incremented. When the indexes reach the end of the array, they wrap around to the beginning. If the `read` and `write` indexes are equal, the buffer is empty. If the `write` index is one slot behind the `read` index, the buffer is full.

Circular buffers are useful for sharing data between interrupts and mainline code. The ISR can write data to the buffer while the mainline code reads from the buffer and sends the data over the UART.

Code Sample 11.4 demonstrates the concept of decimation and Code Sample 11.5 demonstrates the concept of a circular buffer. Both programs send 5000 data samples from the PIC32 to the host computer. Code Sample 11.4 is titled `batch.c` because all decimated data is first stored in an array in RAM, then sent over the UART in one batch. Code Sample 11.5 is called `circ_buf.c` because data is streamed using a circular buffer. The use of a circular buffer (a) allows the buffer to use less RAM than the array in `batch.c` and (b) allows data to be sent immediately, not just in a batch after it is all collected.

Code Sample 11.4 `batch.c`. Storing Data in an ISR and Sending it in a Batch Over the UART.

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

#define DECIMATE 3              // only send every 4th 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) {  // Timer1 ISR operates at 5 kHz
  static int count = 0;         // counter used for decimation
  static int i = 0;             // the data returned from the isr
  ++i;                          // generate the data (we just increment it for now)
  if(count == DECIMATE) {       // skip some data
    count = 0;
    if(curr < NSAMPLES) {
      data_buf[curr] = i;       // queue a number for sending over the UART
```

```
      ++curr;
    }
  }
  ++count;
  IFS0bits.T1IF = 0;            // clear interrupt flag
}

int main(void) {
  int i = 0;
  char buffer[100] = {};
  NU32_Startup();               // cache on, 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 5
  IFS0bits.T1IF = 0;            // clear the interrupt flag
  IEC0bits.T1IE = 1;            // enable the interrupt
  T1CONbits.ON  = 1;            // turn the timer on
  __builtin_enable_interrupts(); // INT step 7: enable interrupts at CPU

  NU32_ReadUART3(buffer, sizeof(buffer)); // wait for the user to press enter
  while(curr !=NSAMPLES) { ; }            // wait for the data to be collected

  sprintf(buffer,"%d\r\n",NSAMPLES);      // send the number of samples that will be sent
  NU32_WriteUART3(buffer);

  for(i = 0; i < NSAMPLES; ++i) {
    sprintf(buffer,"%d\r\n",data_buf[i]);  // send the data to the terminal
    NU32_WriteUART3(buffer);
  }
  return 0;
}
```

Code Sample 11.5 circ_buf.c. Streaming Data from an ISR Over the UART, Using a Circular Buffer.

```
#include "NU32.h" // constants, functions for startup and UART
// 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 unlike 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; // circular buf indexes
static volatile int start = 0;                  // set to start recording

int buffer_empty() {    // return true if the buffer is empty (read = write)
  return read == write;
}

int buffer_full() {     // return true if the buffer is full.
  return (write + 1) % BUFLEN == read;
}
```

```
int buffer_read() {     // reads from current buffer location; assumes buffer not empty
  int val = data_buf[read];
  ++read;               // increments read index
  if(read >= BUFLEN) {  // wraps the read index around if necessary
    read = 0;
  }
  return val;
}

void buffer_write(int data) { // add an element to the buffer.
  if(!buffer_full()) {        // if the buffer is full the data is lost
    data_buf[write] = data;
    ++write;                  // increment the write index and wrap around if necessary
    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, 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 5
  IFS0bits.T1IF = 0;              // clear the interrupt flag
  IEC0bits.T1IE = 1;              // enable the interrupt
  T1CONbits.ON  = 1;              // turn the timer on
  __builtin_enable_interrupts();    // INT step 7: enable interrupts at CPU

  NU32_ReadUART3(msg,sizeof(msg));  // wait for the user to press enter before continuing
  sprintf(msg, "%d\r\n", NSAMPLES); // tell the client how many samples to expect
  NU32_WriteUART3(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, send data over uart
    NU32_WriteUART3(msg);
  }

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

In circ_buf.c, if the circular buffer is full, data is lost. While circ_buf.c is written to send a fixed number of samples, it can be easily modified to stream samples indefinitely. If the UART baud is sufficiently higher than the rate at which data bits are generated in the ISR, lossless data streaming can be performed indefinitely. The circular buffer simply provides some cushion in cases where communication is temporarily delayed or disrupted.

11.3.5 Communication with MATLAB

So far, when we have used the UART to communicate with a computer, we have opened the serial port in a terminal emulator. MATLAB can also open serial ports, allowing communication with and plotting of data from the PIC32. As a first example, we will communicate with talkingPIC.c from MATLAB.

First, load talkingPIC.c onto the PIC32 (see Chapter 1 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 PORT 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 PORT.

% 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 batch.c or circ_buf.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 is defined. This code will plot data generated by either batch.c or circ_buf.c.

Code Sample 11.8 uart_plot.py. Python Code to 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())  #tell the pic to send data. encode converts to a 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 Wireless Communication with an XBee Radio

XBee radios are small, low-power radio transmitters that allow wireless communication over tens of meters, according to the IEEE 802.15.4 standard (Figure 11.3). Each of the two communicating devices connect to an XBee through a UART, and then they can communicate wirelessly as if their UARTs were wired together. For example, two PIC32s could talk to each other using their UART3s using the NU32 library.

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 115,200. The easiest way to configure 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 drop-in replacements, one at each UART, for the wires that would usually connect them.[3]

[3] One caveat occurs at higher baud rates. The XBee generates its baud by dividing an internal clock signal. This clock does not actually achieve a baud of 115,200, however; when set to 115,200 the baud actually is 111,000. 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 111,000 to match the actual baud of the XBee.

Figure 11.3
An XBee 802.15.4 radio. (Image courtesy of Digi International, digi.com.)

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 a UART 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 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 eight-byte hardware FIFOs, one for receiving, one for sending. These FIFOs buffer data in hardware, allowing software to temporarily attend to other tasks while the buffers are being transmitted or filled by received data.
- 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.

11.6 Exercises

1. Plot the waveform for a UART sending the byte 0b11011001, assuming 9600 baud, no parity, 8 data bits, and one stop bit.

2. Write a program that reads characters that you type in your terminal emulator (via UART3), capitalizes them, and returns them to your computer. Rather than processing each character one line at a time, you want a result after each character is pressed; therefore, you cannot use `NU32_WriteUART3` or `NU32_ReadUART3`. For example, if you type 'x' in the terminal emulator, you should see 'X'. You can use the C standard library function `toupper`, defined in `ctype.h`, to convert characters to upper case.

Further Reading

PIC32 family reference manual. Section 21: UART. (2012). Microchip Technology Inc.
XBee/XBee-PRO RF modules (v1.xEx). (2009). Digi International.