Laboratory IV
Universal Asynchronous Receiver Transmitter

Contents

Laboratory IV
Universal Asynchronous Receiver Transmitter

Christopher Vickery
CS-345, Spring 2004

Abstract

We did two projects to investigate serial communication between a Celoxica RC200E FPGA development kit, a keyboard, and an RS-232 port connected to a COM port on a PC. The first project, LCD_Console read characters from a keyboard attached to the RC200E's PS/2 keyboard port and displayed the characters, along with their ASCII codes, on the LCD display of the RC200E. The second project implemented a complete Universal Asynchronous Received Transmitter (UART) on the RC200E's FPGA. Using the UART we were able to simultaneously read characters from the PS/2 port and transmit them to a PC's COM port while reading characters arriving from the COM port and displaying them on the LCD.

Introduction

The RS-232 standard defines a method for transmitting data a character at a time over a serial link between two devices. Two signal wires are used for full-duplex operation, RxD for receiving data bits, and TxD for transmitting data bits. Each signal wire carries one of two voltages, referred to as "mark" and "space." Logically, the mark voltage is a binary 1, and the space voltage is binary 0. Several parameters must be agreed on in order for two systems to use RS-232 to exchange characters successfully:

- Baud Rate. The rate at which bits are transmitted over the serial link. The reciprocal of the baud rate is amount of time each bit occupies the RxD or TxD wire.
- Bits Per Character. Each character may be 5, 6, 7, or 8 bits long.
- Parity. Each character may optionally include a parity bit. If present, this bit may be set for even parity, odd parity, mark, or space.
- Number Of Stop Bits. Each character begins with a transition from mark to space for one bit interval, called the "start bit." Then the data bits and parity bit (if present) are transmitted, and finally the end of the character is marked by a transition from space back to mark for a specified minimum amount of time. This minimum amount of time may be equal to 1.0, 1.5, or 2.0 bit times, which is called the "stop bit(s)."
- Flow Control. Flow control refers to the ability of a receiver to indicate whether it is ready to receive characters or not and the ability of a transmitter to indicate whether it has data to send or not. It can be implemented by using two additional signal wires (besides RxD and TxD) called Clear To Send (CTS) and Ready To Send (RTS), which is referred to as "hardware flow control." Alternatively, special characters can be embedded in the streams of characters being exchanged to provide these functions, a technique called "software flow control." Alternatively, communication can be done without any flow control.

The job of a UART is to serialize for transmission, to deserialize received data, and to check for various errors that might occur. These errors include:

- Parity Error (PE). If parity is used and the value of the parity bit is incorrect, a parity error has occurred. The "correctness" of the parity bit depends on the type of parity checking being used:
    - Mark. The parity bit must be 1.
    - Space. The parity bit must be 0.
    - Even. The sum of the number of ones in the data bits and the parity bit must be 0 modulo 2.
    - Odd. The sum of the number of ones in the data bits and the parity bit must be 1 modulo 2.
- Framing Error (FE). The RxD line must be in the mark state during the stop bit(s). If not, a framing error has occurred.
- Data Overrun (DO). If the UART is ready to produce a second received character before the previous one has been consumed, a data overrun error has occurred.

The Celoxica RC200E development kit provides a Xilinx XC2V1000 FPGA with connections to a PS/2 keyboard port, and LCD display, and an RS-232 DB9 connector, among other things. The interface between the DB9 connector and the FPGA consists of four signal wires (RxD, TxD, CTS, and RTS) connected through a transceiver circuit that translates between RS-232 voltages and the FPGA's logic 1 and 0 voltages. Celoxica provides Platform Abstraction Layer (PAL) macros for reading characters from the PS/2 port, writing characters to the LCD, and reading/writing characters using the RS-232 port. In this laboratory, we wanted to learn to use the PAL macros for reading from the PS/2 port and writing to the LCD console, and then to implement our own UART for communicating with the RS-232 port.

Method

The first project, Keyboard_LCD, used the *PalKeyboardReadASCII()* macro to read characters from the PS/2 keyboard port and then used *PalConsolePutChar()* and *PalConsolePutString()* macros to write each character, along with the hexadecimal representation of its ASCII code, on the LCD. The Handel-C code for the program used, *keyboard_lcd.hcc*, is given in Appendix A.

The second project, *UART* implemented a UART with the following parameters: 38400 baud, no parity, 8 data bits, and no flow control. The baud rate could be changed by adjusting the value of a macro expr "baud_rate" on line 32 of the source code of *uart.hcc* given in Appendix B. The code provides similar mechanisms for adjusting the parity mode and the number of stop bits, but these features were not implemented.

The UART code was organized as three *main()* functions, one to act as a driver controlling the flow of data, a second one to receive serial data from the RxD pin and convert it into an 8-bit value in a register called the RBR, which stands for "receive buffer register". The third *main()* was responsible for transmitting characters over the TxD line as they arrived from a *chan* called THR, which stands for "transmit holding register." After setting everything up, the driver thread was responsible for reading characters from the keyboard port and sending them to the transmitter using the THR *chan*. The driver also consumed characters produced by the receiver, and wrote them to the LCD console.

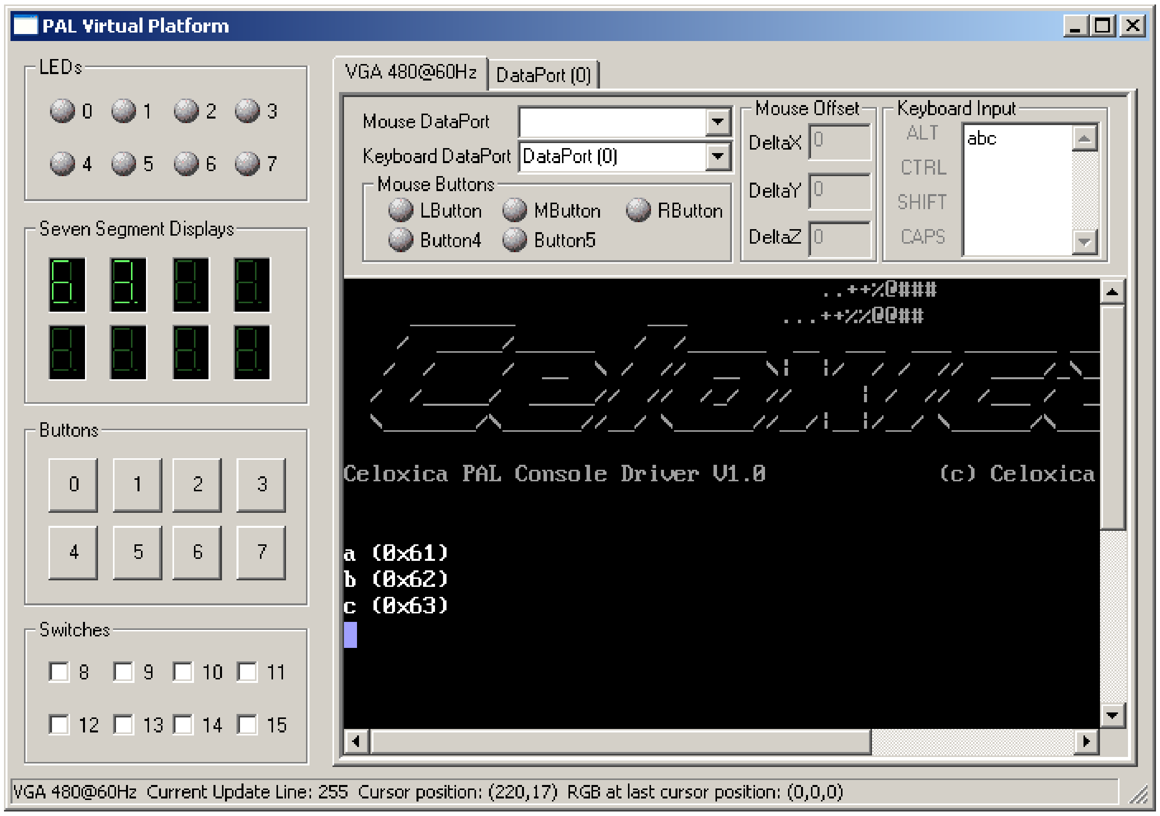
The receiver checked for framing and data overrun errors, and turned on LEDs on the RC200E if these errors occurred. The error conditions could be cleared by pressing the buttons on the side of the RC200E.

The UART was tested by connecting a RC200E to the COM1 port of a PC using a crossover cable that connected pin 2 on the RC200E's DB9 connector (RxData) to pin 3 of the COM1 port's DB9 connector (TxData), and vice-versa. I used the Windows Hyperterm application to send and receive characters on the PC for testing. There was no way to test the DO and FE error handling parts of the code because these events never occurred during final testing of the code.

Results

Keyboard_LCD

The Keyboard_LCD project worked as expected, both for simulation and when downloaded to an RC200E. Typing characters on a keyboard attached to the PS/2 port appeared on the screen, along with the hexadecimal representations of their ASCII codes, one per line. In addition, the ASCII code for each character appeared on the seven segment displays. However, some keyboard characters were not displayed correctly (~, |, and \ ), which seemed to be a "feature" of the *PalKeyboardReadASCII()* macro. Figure 1 shows sample output during simulation.



Mouse DataPort:
Keyboard DataPort: DataPort (0)
DeltaX: 0
DeltaY: 0
DeltaZ: 0
Keyboard Input: abc

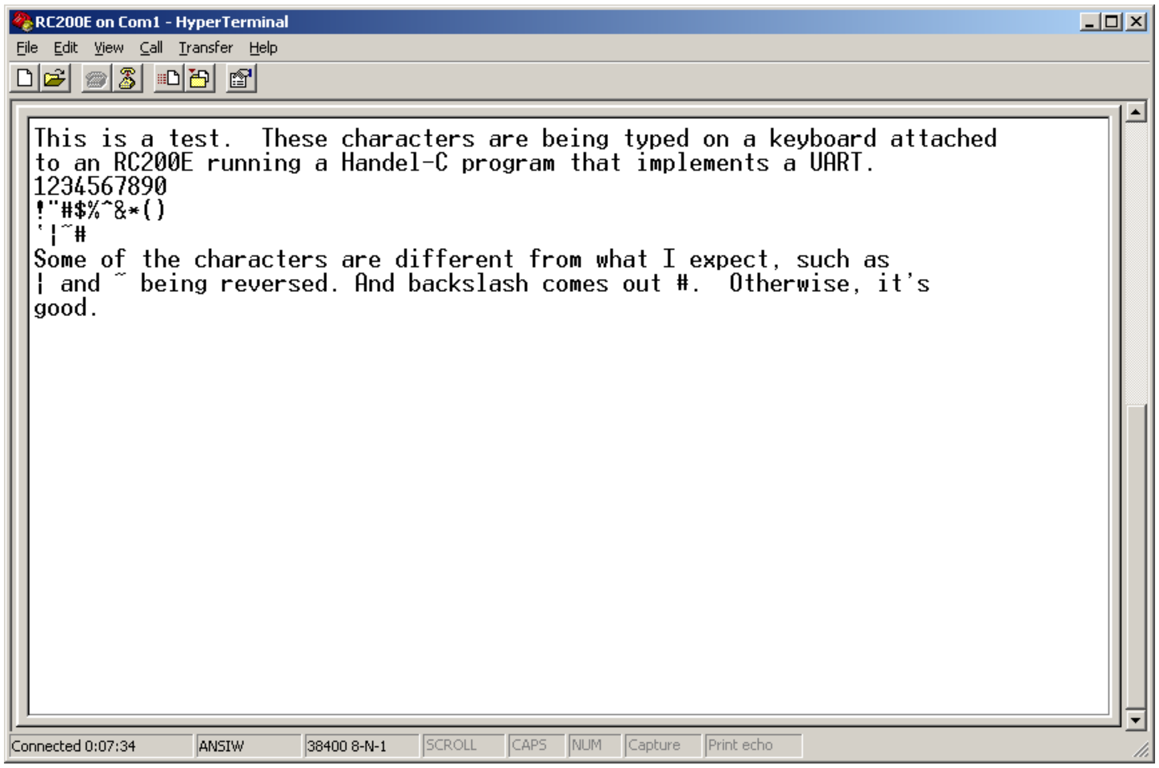
```
Celoxica PAL Console Driver V1.0          (c) Celoxica


a (0x61)
b (0x62)
c (0x63)
```

VGA 480@60Hz Current Update Line: 255 Cursor position: (220,17) RGB at last cursor position: (0,0,0)

Figure 1. Simulation display for the Keyboard_LCD project after typing the letters a, b, and c.

UART

The UART program successfully exchanged characters to the PC's COM1 port. The driver function was able to display characters received from the UART on the console, and characters typed on the PS/2 port's keyboard showed up correctly on the Hyperterm screen. Figure 2 is a screenshot of the Hyperterm application running during testing. The DK2 simulator does not simulate RS-232 I/O without elaborate configuration of the Waveform Analyzer's pattern generator facility, so this application was able to run only on a real RC200E.


RC200E on Com1 - HyperTerminal
File Edit View Call Transfer Help
Connected 0:07:34
ANSIW
38400 8-N-1
SCROLL
CAPS
NUM
Capture
Print echo

```
This is a test.  These characters are being typed on a keyboard attached
to an RC200E running a Handel-C program that implements a UART.
1234567890
!"#$%^&*()
'¦~#
Some of the characters are different from what I expect, such as
¦ and ~ being reversed. And backslash comes out #.  Otherwise, it's
good.
```

Figure 2. The Hyperterm application displaying characters received from the COM1 port, which was connected by a crossover cable to an RC200E.

Discussion

Debugging the UART without the aid of simulation was particularly challenging. When the code wasn't working it was very difficult to locate the problem. The main issue was with the receiver, and I was forced to copy the received data to an expansion header pin so I could look at it with an oscilloscope to see what the data being transmitted actually looked like. In addition to providing experience with the RS-232 serial communication protocol, the project helped emphasize how important it is to have good debugging tools.

Appendix A – keyboard_LCD.hcc

```
//  keyboard_lcd.hcc

/*
 *    Read from RS-232 serial port, write to PAL Console.
 *
 *      C. Vickery
 *      Spring 2004
 */

#define PAL_TARGET_CLOCK_RATE 25175000
#include <stdlib.hch>
#include <pal_master.hch>
#include <pal_keyboard.hch>
#include <pal_console.hch>

//  RC200 Devices
PalConsole  *consolePtr;
PalKeyboard *keyboardPtr;

//  Hexadecimal Conversion Table
static rom hexTab[16] = "0123456789ABCDEF";

//  Output Strings
static unsigned 8 leader[4]   = " (0x";
static unsigned 8 trailer[2]  = ")\n";

//  msec_delay()
//  ------------------------------------------------------------------
/*
 *    Utility for timing delays.
 */
  macro proc msec_delay( msec )
  {
    macro expr ticks = (PAL_ACTUAL_CLOCK_RATE / 1000) * msec;
    unsigned (log2ceil( ticks ) ) count;
    count = ticks;
    while ( count > 0 )
    {
      count--;
    }
  }

//  main()
//  ------------------------------------------------------------------
/*
 *    Read characters from serial port, and write them to the PAL
 *    console.
 */
  void main( void )
  {
    unsigned 8  ch; //  Keyboard input

    //  Check platform devices available.
    PalVersionRequire( 1, 0 );
    PalVideoOutRequire( 1 );
    PalPS2PortRequire( 2 );

    par
    {
      //  Initialize all devices
      PalKeyboardRun( &keyboardPtr, PalPS2PortCT( 1 ),
                                              PAL_ACTUAL_CLOCK_RATE );
      PalConsoleRun(  &consolePtr, PAL_CONSOLE_FONT_NORMAL,
                        PalVideoOutOptimalCT( PAL_ACTUAL_CLOCK_RATE ),
                                              PAL_ACTUAL_CLOCK_RATE );
      seq
      {
        par
        {
          PalKeyboardEnable( keyboardPtr );
          PalSevenSegEnable( PalSevenSegCT( 0 ) );
          PalSevenSegEnable( PalSevenSegCT( 1 ) );
          PalConsoleEnable( consolePtr );
        }

        //  Read from keyboard, write to console
        while (1)
        {
          PalKeyboardReadASCII( keyboardPtr, &ch );
          par
          {
            //  Display the ASCII code on the seven segment displays
            PalSevenSegWriteDigit( PalSevenSegCT( 1 ), ch <- 4, 1 );
            PalSevenSegWriteDigit( PalSevenSegCT( 0 ), ch \\ 4, 0 );
            //  Display the character and its ASCII code on the LCD
            seq
            {
              PalConsolePutChar(    consolePtr, ch );
              PalConsolePutString(  consolePtr, leader );
              PalConsolePutChar(    consolePtr, hexTab[ch \\ 4] );
              PalConsolePutChar(    consolePtr, hexTab[ch <- 4] );
              PalConsolePutString(  consolePtr, trailer );
            }
          }
          //  Allow time to look at the seven segment displays
          msec_delay( 250 );
        }
      }
    }
  }
```

Appendix B – UART.hcc

```
//  uart.hcc

/*  Implements a UART.
 *
 *    Goal:             RS-232 I/O through the RC200E serial port.
 *    Parameters:       Various baud rates, 8 data bits, no parity, no
 *                      flow control.
 *    Test Environment: Read chars from serial port, write to
 *                      lcd console.  Read chars from keyboard,
 *                      write to serial port.
 *
 *    C. Vickery
 *    Spring 2004
 */

#define PAL_TARGET_CLOCK_RATE 25175000

#include <stdlib.hch>
#include <pal_master.hch>
#include <pal_console.hch>
#include <pal_keyboard.hch>

macro expr CLOCK = PAL_ACTUAL_CLOCK_RATE;

//  Manifest Constants
#define BAUD_38400  38400
#define BAUD_9600    9600
#define PARITY_NONE   000
#define PARITY_MARK   001
#define PARITY_SPACE  010
#define PARITY_ODD    011
#define PARITY_EVEN   100

//  RS-232 Parameters
macro expr baud_rate      = BAUD_38400;
macro expr bits_per_char  = 8;
macro expr parity         = PARITY_NONE;
macro expr stop_bits      = 1;

//  Logic levels for two states of the serial line
macro expr mark = 1;
macro expr space = 1 - mark;

//  Serial data and status values
static unsigned 1 rx_data   = mark; //  serial data in
static unsigned 1 tx_data   = mark; //  serial data out
static unsigned 1 FE        = 0;    //  framing_error
static unsigned 1 DO        = 0;    //  data_overrun
static unsigned 1 rbrFull   = 0;
static unsigned 8 RBR       = 0;    //  receive_buffer_register
chan   unsigned 8 THR;              //  transmit_holding_register

/*  Alternate to using a chan for THR   */
//static unsigned 8 THR       = 0;
```

```
//static unsigned 1 thrEmpty  = 1;

//  Connections to Tx and Rx data pins
interface bus_out() serial_out( unsigned 1 data = tx_data )
  with
  {
    data = { "V20" }
  };

interface bus_in( unsigned 1 data) serial_in()
  with
  {
    data = { "U20" }
  };


//  Copy i/o data to expansion header for oscilloscope monitoring
interface bus_out() expansion_out( unsigned 2 data = tx_data@rx_data )
  with
  {
    data = { "M2", "N2" }
  };


//  Indicators for framing errors and data overrun errors.
macro expr  led_0 = PalLEDCT( 0 );
macro expr  led_1 = PalLEDCT( 1 );


//  Handles for RC200's PS/2 keyboard and LCD console.
PalKeyboard *keybd;
PalConsole  *console;


//  Function Prototypes
//  ------------------------------------------------------------------
void keybd_to_port( void );
void port_to_console( void );


//  wait_half_bit()
//  ------------------------------------------------------------------
/*
 *    Macro proc to delay for some multiple of half bits.
 */
  macro expr ticks_per_bit = CLOCK / baud_rate;
  macro proc wait_half_bit( num )
  {
    macro expr ticks = num * (ticks_per_bit / 2);
    unsigned (log2ceil( ticks )) count;
    count = ticks - 1;
    while ( count > 0 )
      count--;
  }


//  main() - Driver
//  ----------------------------------------------------------------------
/*
 *    Check dependencies, initialize console and keyboard, and launch
 *    functions to read and write the serial port.
 */
  void main( void )
  {
    //  Check dependencies
    PalPS2PortRequire( 2 );
    PalVideoOutRequire( 1 );
    PalSevenSegRequire( 2 );
    PalSwitchRequire( 2 );
    PalLEDRequire( 2 );

    //  Initialize console and keyboard.
    par
    {
      PalConsoleRun( &console, PAL_CONSOLE_FONT_NORMAL,
                              PalVideoOutOptimalCT( CLOCK ), CLOCK  );
      PalKeyboardRun( &keybd, PalPS2PortCT( 1 ), CLOCK );
      seq
      {
        par
        {
          PalConsoleEnable( console );
          PalKeyboardEnable( keybd );
          //  Seven Segment displays used for debugging
          PalSevenSegEnable( PalSevenSegCT( 0 ) );
          PalSevenSegEnable( PalSevenSegCT( 1 ) );
        }

        //  Launch the driver's processing functions
        par
        {
          keybd_to_port();
          port_to_console();
        }
      }
    }
  }


//  keybd_to_port()
//  ----------------------------------------------------------------------
/*
 *    Read from Keyboard, write to RS-232 port
 */
  void
  keybd_to_port( void )
  {
    unsigned 8 keybdChar;

    while ( 1 )
    {
      //  Get a char, send it to THR channel
      PalKeyboardReadASCII( keybd, &keybdChar );
      THR ! keybdChar;
  /*  Alternate implementation: Use a 1-bit variable thrEmpty to
   *  synchronize interaction between this code and the transmitter
   *  through a register named THR.  This is like typical UART
   *  integrated circuits.
   */
    //    do
    //    {
    //      delay;
    //    } while ( 0 == thrEmpty );
    //    par
    //    {
    //      THR = keybdChar;
    //      thrEmpty = 0;
    //    }
    //
      par
      {
        //  Debugging: display typed chars on seven segment displays.
        PalSevenSegWriteDigit( PalSevenSeg( 0 ), keybdChar[7:4], 1 );
        PalSevenSegWriteDigit( PalSevenSeg( 1 ), keybdChar[3:0], 1 );
      }
    }
  }


//  port_to_console()
//  ------------------------------------------------------------------
/*
 *    Read from RS-232 port, write to LCD console.  Manage framing
 *    and data overrun errors.
 */
  void
  port_to_console( void )
  {
    unsigned 1 clearFE, clearDO;  //  Use switches to clear errors

    par
    {
      //  Read from port
      while ( 1 )
      {
        //  Wait for char
        do
        {
          delay;
        } while ( rbrFull == 0 );

        par
        {
          //  Write char to console
          par
          {
            rbrFull = 0;
            PalConsolePutChar( console, RBR );
          }
        }
      }

      //  Use switch 0 to clear framing errors
      while ( 1 )
      {
        do
        {
          PalSwitchRead( PalSwitchCT( 0 ), &clearFE );
        } while ( clearFE == 0 );
        FE = 0;
      }

      //  Use switch 1 to clear data overruns
      while ( 1 )
      {
        do
        {
          PalSwitchRead( PalSwitchCT( 1 ), &clearDO );
        } while ( clearDO == 0 );
        DO = 0;
      }
    }
  }

//  main() - UART Transmitter
//  -----------------------------------------------------------------
/*
 *    Write bytes from THR to serial port's Tx line (Pin V20).
 *
 *    Algorithm:
 *      Wait for char to arrive in THR
 *      Copy it to transmitter and signal THR empty.
 *      Write a start bit
 *      Write data bits
 *      Write parity bit if needed
 *      Write stop bit(s)
 *
 */
  void main( void )
  {
    unsigned 4  i;
    unsigned 8  transmitter;
    while ( 1 )
    {
      //  Wait for char
      THR ? transmitter;

/*  Alternate implementation    */
//    do
//    {
//      delay;
//    } while ( 1 == thrEmpty );
//
//    //  Copy to transmitter and signal THR empty
//    par
//    {
//      transmitter = THR;
//      thrEmpty = 1;
//    }

      par
      {
        //  Debugging: Write received chars on seven seg displays
        PalSevenSegWriteDigit( PalSevenSeg( 0 ), transmitter[7:4], 0 );
        PalSevenSegWriteDigit( PalSevenSeg( 1 ), transmitter[3:0], 0 );
      }

      //  Write start bit, data bits, parity bit, stop bits.
      tx_data = space;              //  Start bit
      wait_half_bit( 2 );
      for ( i = 0; i < bits_per_char; i++ )
      {
        par
        {
          tx_data = transmitter[0]; // Data bit
          transmitter = transmitter >> 1;
        }
        wait_half_bit( 2 );
      }
                                    //  Parity bit
      tx_data = mark;               //  Stop bit(s)
      wait_half_bit( 2 );

    }
  }

//  main() - UART Receiver
//  ------------------------------------------------------------------
/*
 *    Read bytes by monitoring the serial port's Rx line (Pin U20).
 *
 *    Algorithm:
 *      Wait for start of start bit (Rx goes from mark to space)
 *      Wait 1/2 bit interval
 *      Read start bit (must be space)
 *      8 times: Wait 1 bit interval, read data bit
 *      Wait one bit interval, read stop bit (must be mark).
 *      Write char to RBR.
 *
 *      Turn on framing error and data overrun indicators when those
 *      conditions are detected.
 */
  void main ( void )
  {
    unsigned 4 i;
    unsigned 8 receiver;

    par
    {
      //  Sample serial input line on each clock tick
      //  Display framing errors and data overruns on the LEDs
      while ( 1 )
      {
        par
        {
          rx_data = serial_in.data;
          PalLEDWrite( PalLEDCT( 0 ), FE );
          PalLEDWrite( PalLEDCT( 1 ), DO );
        }
      }

      //  Assemble characters and transfer them to RBR
      while ( 1 )
      {
        //  Wait for beginning of start bit
        do
        {
          delay;
        }
        while ( rx_data == mark );

        //  Check middle of start bit
        wait_half_bit( 1 );
        if ( rx_data != space )
        {
          //  Glitch ... wait for start of another start bit
          continue;
        }

        //  Collect the 8 data bits, lsb first
        for (i = 0; i < 8; i++ )
        {
          wait_half_bit( 2 );
          receiver = (rx_data@receiver) \\ 1;
        }

        //  Check stop bit
        wait_half_bit( 2 );
        if ( rx_data != mark )
        {
          FE = 1;
        }

        par
        {
          //  Dump receiver into RBR; check for overrun
          RBR = receiver;
          if (rbrFull )
          {
            DO = 1;
          }
          else
          {
            rbrFull = 1;
          }
        }
      }
    }
  }
```