UNIVERSAL ASYNCHRONOUS RECEIVER TRANSMITTER.

(LABORATORY IV)

3/21/2004

Sergey Averchenkov

CS345, Queens College, N.Y.

INTRODUCTION:

The purpose of this lab was to study standard RS-232 serial communication between RC200E device and the PC. A regular desktop computer equipped with serial ports has UART, Universal Asynchronous Receiver Transmitter. This device serializes the data coming from CPU and transmits it bit by bit over the standard RC-232 cable to any other device. It is also responsible for receiving data coming in and converting transmitting it to the CPU. While all PCs are generally equipped with UARTs, the RC200E used in this lab lacks such device. Our task was to create a program that would simulate the behavior of a UART, which would allow us to receive and transmit data from or to the PC.

The RC200E provides a "DB9" connector with pins numbered 2, 3, 7, and 8 connected, respectively, to pins T19, U20, U19, and V20 of the FPGA through a transceiver circuit that translates between RS-232 voltage levels used for serial I/O and the voltage levels used for pin I/O by the FPGA. Using pin V20 for transmitting data and pin U20 for receiving data, we were able to simulate UART behavior in the FPGA.

This project provided us with deeper understanding of serial communications used in many common devices such as external modems or many USB devices that are commonly found on any PC. By examining the serial communication standards, we learned the protocol used in data transfer and issues related to data validation and error checking.

Project Specifications:

This project is to provide a serial connection between the FPGA and a PC through the RS-232 connector on the RC200E and one of the COM ports on a PC. Writing characters to the COM port on the PC is to cause them to display on the LCD screen of the RC200E. At the same time, typing characters on a keyboard attached to the PS/2 keyboard port of the RC200 is to cause them to display on the screen of the PC.

METHOD:

This project can be broken into seven main parts as follows:

Driver – main():

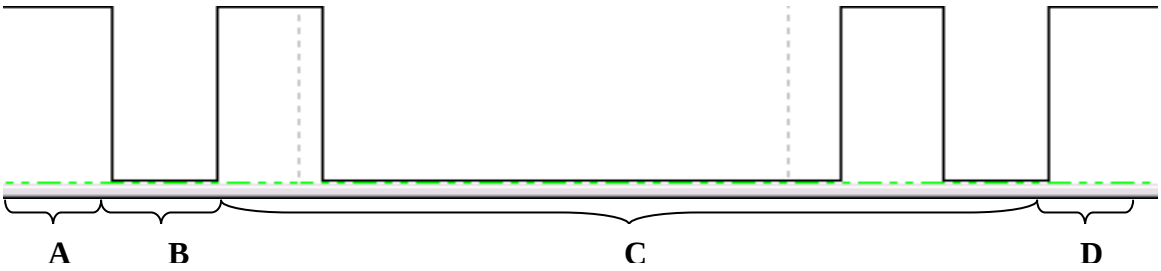
This thread enables keyboard and LCD screen and starts six concurrent threads that perform all the required program functionality.

RxRun() – Data receiver.
TxRun() – Data Transmitter.
ConsoleRun() – Console output.
KeyboardRun() – Keyboard input.
DoMonitorRun() – Data overflow monitor.
FEMonitorRun() – Framing error monitor.

Receiver – RxRun():
This thread monitors the voltage on the U20 pin on the RC200E device according to RS-232 UART specification. The receiver reads the values from the input pin and stores them in the local buffer. Low-order bits are read first. When all bits have been read, the receiver updates the RBR (Receiver Buffer Register) and sets the value of rbrFull flag to TRUE.

RS232 is configured as follows:

- Bits Per Second: 9600
- Data Bits: 8
- Parity: None
- Stop Bits: 1
- Flow Control: None



A – **Default voltage.** When no data is being transmitted, the line remains in MARK position.

B – **Starting Bit.** When data is available for transmission, voltage drops from MARK to SPACE for the period of one pulse.

C - **Data Bits.** Data bits are transmitted in these pulses. Low-order bits arrive first.

D - **Stopping Bit.** Voltage goes back to MARK position for the period specified to by both parties involved in communication. (The period chosen for this project is 1 pulse, but acceptable values are usually 1, 1.5, or 2 pulses.)

Certain error conditions may come up in data transfer:

If starting bit is not the SPACE position, then the receiver considers it to be a glitch on the line and simply ignores by looking for the beginning of the new starting bit.

If the stopping bit is not in the MARK position, then Framing Error (FE) occurs. The receiver sets the FE bit to TRUE and stops updating the RBR while FE remains TRUE.

The receiver checks the value of the rbrFull flag before updating the RBR buffer. If the value of rbrFull is TRUE, then Data Overrun (DO) error

occurs. The receiver sets the DO bit to TRUE and stops updating the buffer while DO remains true.

Transmitter – TxRun():
This thread monitors the state of the thrFull flag, and if the flag is true, it copies the value of the THR (Transmitter Holding Register) buffer, sets the flag thrFull to FALSE, and transmits the copy buffer bit by bit to the V20 pin on the RC200E device. Low-order bit is transmitted first.

Console Output – ConsoleRun():
This thread monitors the state of the rbrFull flag, and if the flag is true, it outputs the value of the RBR (Receiver Holding Register) buffer to the LCD screen and sets the flag rbrFull to FALSE.

Keyboard Input – KeyboardRun():
This thread monitors the keyboard, updates the RBR (Receiver Holding Register) buffer with the ASCII value of the key pressed, and sets thrFull flag to TRUE once the buffer has been updated.

Data Overflow Flag Monitor – DOMonitorRun():
This thread monitors the value of one of the switches on RC200E device. If the switch is pressed, then it sets the DO (Data Overrun) flag to FALSE. Concurrently, this thread turns on one of the LEDs when the value of the DO flag is TRUE.

Framing Error Flag Monitor – FEMonitorRun():
This thread monitors the value of one of the switches on RC200E device. If the switch is pressed, then it sets the FE (Framing Error) flag to FALSE. Concurrently, this thread turns on one of the LEDs when the value of the FE flag is TRUE.

Software simulation procedure:
Waveform Analyzer was used to verify the correctness of the code. The project was set up as follows:

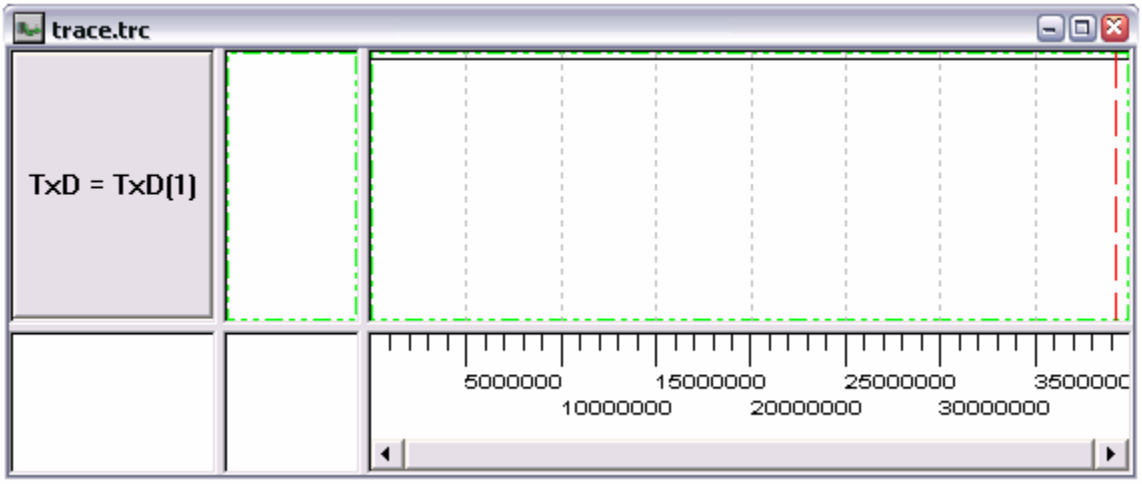
Software simulation clock:

```
#define PAL_ACTUAL_CLOCK_RATE 25175000
set clock = external "P1"
  with
  {
    extlib  = "DKSync.dll",
    extinst = "1000",       // Period of 1MHz simulated clock
    extfunc = "DKSyncGetSet"
  };
```

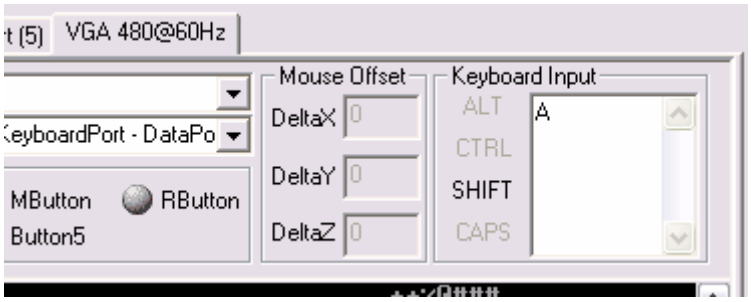
Waveform Analyzer Trace:

Trace is used to check the value output on a pin, which makes it perfect for testing our transmitter. It was set up using these parameters:
Clock Period: 1000
Number of points: 40000
A new trace was created as usual using "TxD(1)" expression.
(See **Appendix A - Source Code** for more details.)
After the trace was created, Handel-C project was built and ran using standard "Simulation" mode. Waveform Analyzer tracing was enabled by clicking "Run" button on the toolbar.

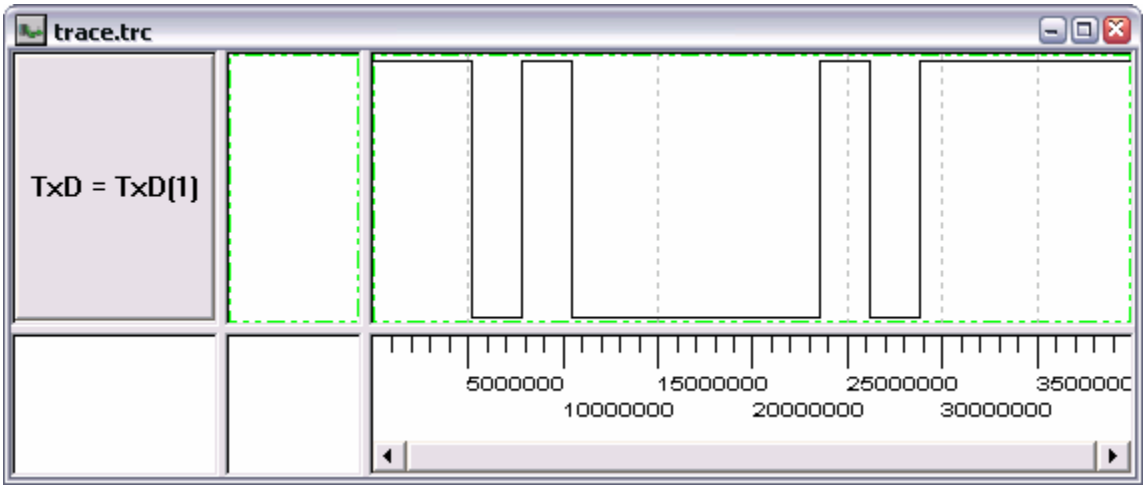
Waveform analyzer started capturing the input from the simulated output pin. By default our program outputs MARK voltage on the output pin (according to the RS232 specification). The output shown in Waveform Analyzer was as expected:



In order to capture the output from the pin, we needed to provide keyboard input for the PAL simulator. It is done by typing characters in "Keyboard Input" textbox in VGA tab of the Pal Virtual Platform simulation program.



Waveform analyzer captured the following output:

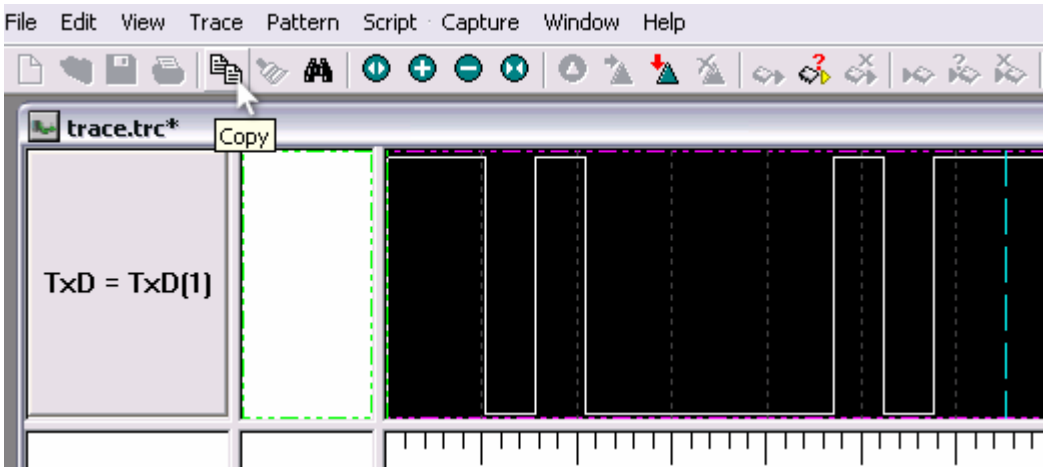


This corresponds exactly to the decimal value 65, which is a capital letter ‘A’. Both stopping and starting bits were present and in correct position.

Waveform Analyzer Pattern:

In order to test the receiver, it was needed to generate a bit pattern according to the required timing and bit values and connect that pattern to the input pin on the simulator. It is a rather complicated task if done using PGL (Pattern generation Language) provided with the Waveform Analyzer application. Instead, we chose to use a simple workaround and use the value of the “Trace” output as the input to the “Pattern”. The procedure is as follows:

Using the steps above, a bit pattern was generated in the “Trace” window. Waveform Analyzer capture was STOPPED. (PAL Simulation was left running.) The pattern was selected using the mouse and copied it into the clipboard.



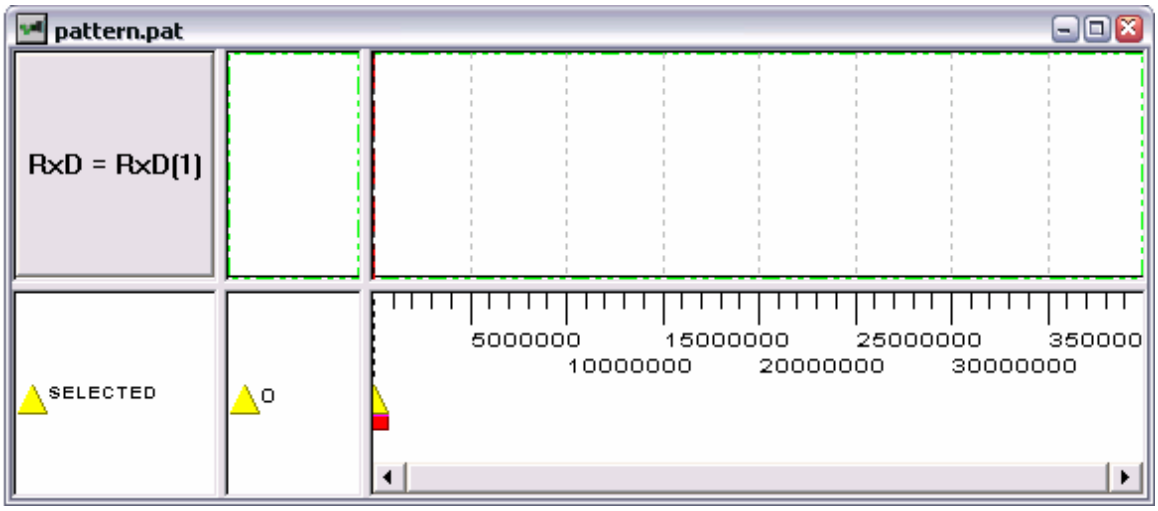
A Pattern was created to generate a series of pulses for the PAL simulation. It was set up as follows:

Clock Period: 1000

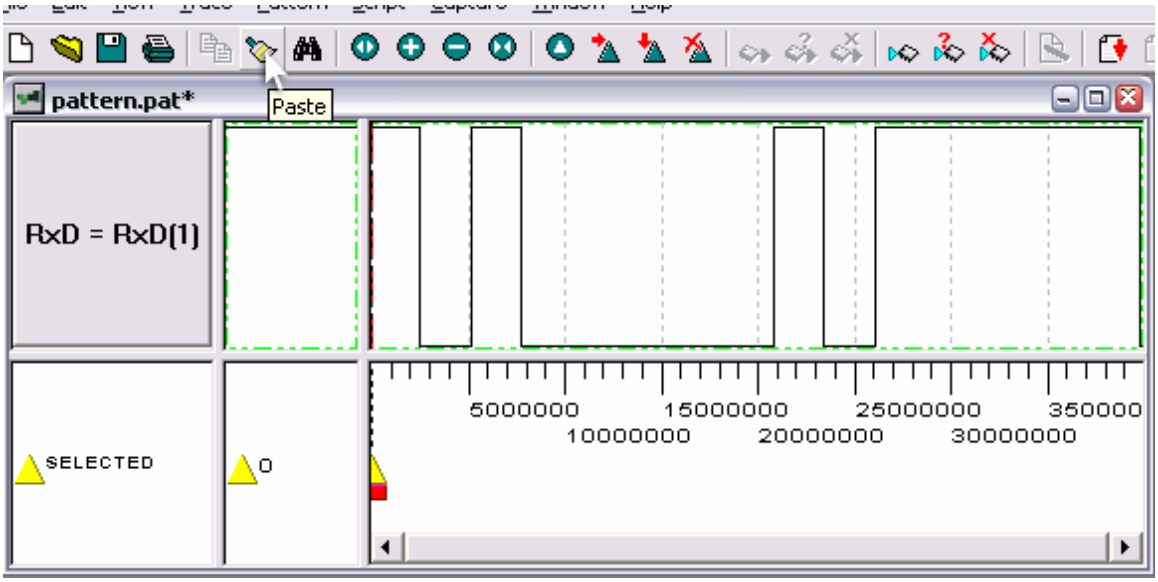
Number of points: 40000

Source: “1;”
Destination: “RxD(1)”

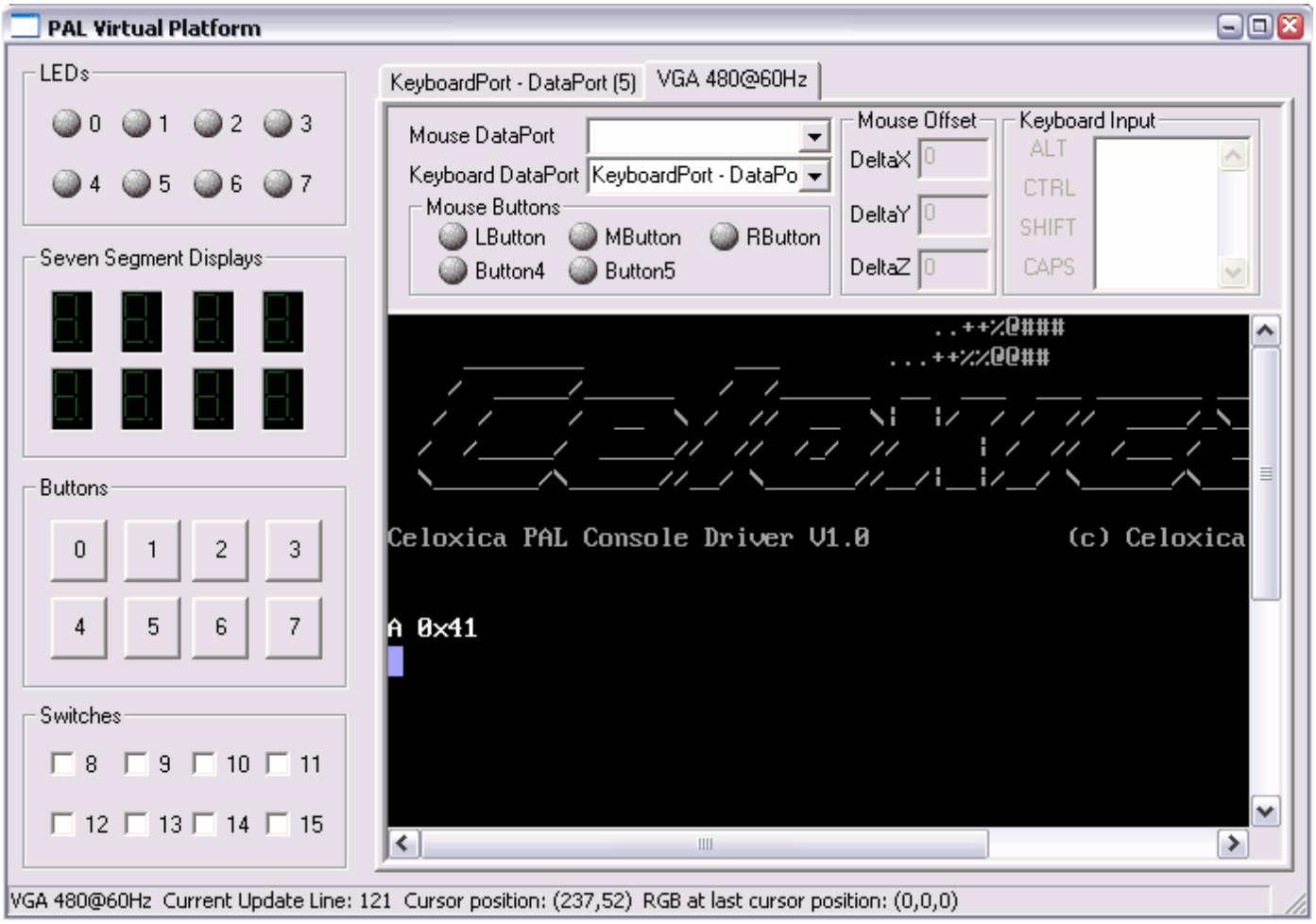
A new marker was created in the Pattern window and moved to the time position zero.



With Pattern window selected, the pattern copied from the Trace window was pasted.



This pattern produced the following output in the PAL simulated console after it was executed by pressing “Go” button on the Wave Analyzer toolbar:



The output was exactly the same as the input we provided for the trace window in the first step of the procedure.

To even further analyze the correctness of the algorithm, it is possible to use breakpoints in the source code in conjunction with the Waveform Analyzer pattern to advance the execution one step at a time checking values of variables at each step.

The major issue with this procedure is that the pattern can only be sent once. Once the pattern is sent, it disappears from the Waveform Analyzer Pattern window. It is, however, possible to simply paste several copies of the pattern at different time locations by moving the marker and pasting it several times.

Another problem is inability to modify the pattern without generating a new test pattern on the output of the transmitter. Waveform Analyzer provides a way to save a pattern to an ASCII text file, but for some unknown reason (I suspect it is due to the lack of user permissions) it is unable to load that pattern back into the program, which considerably complicates the procedure.

RESULTS:

The application written in this lab behaved according to the specifications. Data transmission to and from RC200E device was successful. Software emulation using waveform analyzer also verified the correctness of the program.

Appendix A: Source Code

```
// uart.hcc
/*****************************************************************
* Project   :    RS232 Communication                              *
* Date      :    20 Mar 2004                                      *
* File      :    uart.hcc                                         *
* Author    :    Sergey Averchenkov                               *
*****************************************************************
* Desc      :    This project provides a serial connection        *
*           :    between the FPGA and a PC through the RS-232     *
*           :    connector on the RC200E and one of the COM       *
*           :    ports on a PC. Characters received on the COM    *
*           :    port on the RC200E are displayed on the LCD      *
*           :    screen of the RC200E. At the same time,          *
*           :    characters typed on a keyboard attached to the   *
*           :    PS/2 keyboard port of the RC200 are displayed    *
*           :    on the screen of the PC.                         *
*****************************************************************/

#define MARK 1
#define SPACE 0

#if (defined USE_RC200 || USE_RC200E)
  #define PAL_TARGET_CLOCK_RATE 25175000
  #define SW0 0
  #define SW1 1
  #define NUM_SWITCHES 2
#else
  // Use 1Mhz clock for wave analyzer simulation.
  #define PAL_ACTUAL_CLOCK_RATE 25175000
  //#define PAL_ACTUAL_CLOCK_RATE 1000000
  set clock = external "P1"
    with
    {
      extlib  = "DKSync.dll",
      extinst = "1000",       // Period of 1MHz simulated clock
      extfunc = "DKSyncGetSet"
    };
  #define SW0 8
  #define SW1 9
  #define NUM_SWITCHES 10
#endif

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

unsigned 1 TxD = 1; // Set transmit to MARK

interface bus_in(unsigned 1 RxD) InBus()
    with
    {
  #if (defined USE_RC200 || USE_RC200E)
      data  = { "U20" }
  #else
      extlib  = "DKConnect.dll",
      extinst = "RxD(1)",
```

```
      extfunc = "DKConnectGetSet"
  #endif
    };

interface bus_out() OutBus(unsigned 1 out = TxD)
    with
    {
  #if (defined USE_RC200 || USE_RC200E)
      data  = { "V20" }
  #else
      extlib  = "DKConnect.dll",
      extinst = "TxD(1)",
      extfunc = "DKConnectGetSet"
  #endif
    };

static macro expr ClockRate = PAL_ACTUAL_CLOCK_RATE;
static macro proc microsec_delay(msec);
macro proc delay_FullPulse();
macro proc delay_HalfPulse();
macro proc KeyboardRun(KeyboardPtr);
macro proc ConsoleRun(ConsolePtr);
macro proc DOMonitorRun();
macro proc FEMonitorRun();
macro proc RxRun();
macro proc TxRun();

macro expr BIT_RATE = 9600;
macro expr DATA_BITS = 8;

unsigned (DATA_BITS) RBR = 0;
unsigned (DATA_BITS) THR = 0;

unsigned 1 DO = 0;
unsigned 1 FE = 0;
unsigned 1 rbrFull = 0;
unsigned 1 thrFull = 0;

// main()
// ------------------------------------------------------------
/*
 *    Main entry of the program. Runs all threads.
 */
void main (void)
{
  PalConsole *ConsolePtr;
  PalKeyboard *KeyboardPtr;

  PalVersionRequire(1, 0);
  PalVideoOutRequire(1);
  PalPS2PortRequire(2);

  par
  {
    PalConsoleRun(&ConsolePtr, PAL_CONSOLE_FONT_NORMAL,
      PalVideoOutOptimalCT (ClockRate), ClockRate);
    PalKeyboardRun(&KeyboardPtr, PalPS2PortCT(1), ClockRate);
    seq
    {
```

```
      par
      {
        PalConsoleEnable(ConsolePtr);
        PalKeyboardEnable(KeyboardPtr);
      }
      par
      {
        TxRun();
        RxRun();
        DOMonitorRun();
        FEMonitorRun();
        KeyboardRun(KeyboardPtr);
        ConsoleRun(ConsolePtr);
      }
    }
  }
}
// -------------------------------------------------------------


// -------------------------------------------------------------
// TxRun()
// -------------------------------------------------------------
/*
 *    This thread monitors the InBus.RxD interface according to
 *    RS232 serial communcation protocol. It reads the value
 *    from the interface and updates the RHR buffer.
 *
 *    Exceptions:
 *      - DO is set to TRUE if Data Overrun occurs.
 *        Data Overrun occurs when the receiver tries to
 *        update the buffer that is full.
 *        RHR buffer is not updated while DO is TRUE
 *
 *      - FE is set to TRUE if Framing Error occurs.
 *        Framing Error occurs when the stopping bit is not in
 *        the MARK position during bit transfer.
 *        RHR buffer is not updated while FE is TRUE
 *
 */
macro proc RxRun()
{
  unsigned (DATA_BITS) RxBuffer;
  unsigned 4 i;

  while(1)
  {
    // Started in SPACE. Cable not connected?
    while(InBus.RxD == SPACE) { delay; }
    // MARK is default. Waiting for a start bit.
    while(InBus.RxD == MARK) { delay; }

    // Move in the middle of the pulse
    delay_HalfPulse();

    // Glitch on the line? Expected: SPACE in the
    // middle of the start bit.
    // Restart from the beginning.
    if(InBus.RxD != SPACE) { continue; }
```

```
    delay_FullPulse();

    // Reset the buffer. Ready for receiving bits.
    RxBuffer = 0;
    for(i=0; i < DATA_BITS; i++)
    {
      // Receive each bit and store it int he appropriate
      // pisition in the buffer
      RxBuffer >>= 1;
      RxBuffer = (InBus.RxD @ RxBuffer<-7);
      delay_FullPulse();
    }

    // Stopping bit.
    if(FE || (InBus.RxD != MARK)) // Framing error?
    {
      FE = TRUE;
    }
    else if(DO || rbrFull)
    {
      DO = TRUE;
    }
    else // neither Data Overrun (DO) nor
         // Framing Errors (FE) have occured.
    {
      RBR = RxBuffer; // Copy the buffer
      rbrFull = TRUE;
    }
    // Shift back into the beginning of the pulse
    delay_HalfPulse();
  }
}
// -------------------------------------------------------------


// -------------------------------------------------------------
// TxRun()
// -------------------------------------------------------------
/*
 *    This thread monitors the THR buffer and transmits the
 *    value of the buffer on TxD interface if the buffer is full.
 *    The value is transmitted bit by bit low-order bits first.
 *    This method does not return.
 */
macro proc TxRun()
{
  unsigned (DATA_BITS) TxBuffer;
  unsigned 4 i;

  while(1)
  {
    // Wait for the signal. Data ready to be sent?
    while(!thrFull) { delay; }
    par
    {
      // Make a local copy of the buffer.
      TxBuffer = THR;
      thrFull = 0;
```

```
    }
    // TxD rests at MARK voltage when no data
    // is being transfered. Move to SPACE to indicate
    // the beginning of the start bit.
    TxD = SPACE;
    delay_FullPulse();

    // Send in the bits. Low order bits go in first.
    for(i=0; i < DATA_BITS; i++)
    {
      par
      {
        TxD = TxBuffer[0];
        TxBuffer >>= 1;
      }
      delay_FullPulse();
    }
    // Move the voltage back to MARK for the stopping bit.
    TxD = MARK;
    // Delay for the period of the stopping bit.
    delay_FullPulse();
  }
}
// -------------------------------------------------------------


// -------------------------------------------------------------
// FEMonitorRun()
// -------------------------------------------------------------
/*
 *    This thread monitors the state of one of the switches,
 *    resets the value of the FE variable if the switch was
 *    pressed, and lits the LED if FE is true.
 *    This method does not return.
 */
macro proc FEMonitorRun()
{
  unsigned 1 switchOnePressed;
  while(1)
  {
    // Read the state of the switch and update the LED
    // based on the value of the FE variable.
    par
    {
      PalSwitchRead(PalSwitchCT(SW1), &switchOnePressed);
      PalLEDWrite(PalLEDCT(1), FE);
    }

    // Reset FE to zero if switch was pressed.
    if(switchOnePressed)
    {
      FE = 0;
    }
  }
}
// -------------------------------------------------------------


// -------------------------------------------------------------
```

```
// DOMonitorRun()
// -------------------------------------------------------------
/*
 *    This thread monitors the state of one of the switches,
 *    resets the value of the DO variable if the switch was
 *    pressed, and lits the LED if DO is true.
 *    This method does not return.
 */
macro proc DOMonitorRun()
{
  unsigned 1 switchZeroPressed;
  while(1)
  {
    // Read the state of the switch and update the LED
    // based on the value of the DO variable.
    par
    {
      PalSwitchRead(PalSwitchCT(SW0), &switchZeroPressed);
      PalLEDWrite(PalLEDCT(0), DO);
    }

    // Reset DO to zero if switch was pressed
    if(switchZeroPressed)
    {
      DO = 0;
    }
  }
}
// -------------------------------------------------------------


// -------------------------------------------------------------
// ConsoleRun()
// -------------------------------------------------------------
/*
 *    This thread outputs the value of the RBR buffer to the
 *    LCD screen if the buffer is full. This method does not
 *    return.
 */
macro proc ConsoleRun(ConsolePtr)
{
  static rom unsigned 8 hexTab[] = "0123456789ABCDEF";
  while(1)
  {
    // Wait for the signal. New data ready to be displayed?
    while(!rbrFull) { delay; }

    // Output the value to the screen in the
    // form: "X 0x##"
    PalConsolePutChar(ConsolePtr, RBR);
    PalConsolePutChar(ConsolePtr, ' ');
    PalConsolePutChar(ConsolePtr, '0');
    PalConsolePutChar(ConsolePtr, 'x');
    PalConsolePutChar(ConsolePtr, hexTab[RBR[7:4]]);
    PalConsolePutChar(ConsolePtr, hexTab[RBR[3:0]]);
    PalConsolePutChar(ConsolePtr, '\n');
    // Value was output to the console.
    // Notify the receiver that new data may be sent.
    rbrFull = 0;
```

```
  }
}
// -------------------------------------------------------------


// -------------------------------------------------------------
// KeyboardRun()
// -------------------------------------------------------------
/*
 *    This thread reads the keyboard and updates the THR buffer
 *    if the buffer is empty. This method does not return.
 */
macro proc KeyboardRun(KeyboardPtr)
{
  while(1)
  {
    // Wait for the signal. Is buffer empty?
    while(thrFull) { delay; }
    par
    {
      PalKeyboardReadASCII(KeyboardPtr, &THR);
      thrFull = 1;
    }
  }
}
// -------------------------------------------------------------


// -------------------------------------------------------------
// delay_HalfPulse()
// -------------------------------------------------------------
/*
 *    Pauses the execution for one half of a pulse based on the
 *    current bit rate.
 */
macro proc delay_HalfPulse()
{
  microsec_delay(1000000 / (BIT_RATE * 2));
}
// -------------------------------------------------------------


// -------------------------------------------------------------
// delay_FullPulse()
// -------------------------------------------------------------
/*
 *    Pauses the execution for one full pulse based on the
 *    current bit rate.
 */
macro proc delay_FullPulse()
{
  microsec_delay(1000000 / BIT_RATE);
}
// -------------------------------------------------------------


// -------------------------------------------------------------
// microsec_delay()
// -------------------------------------------------------------
```

```
/*
 *    Pauses the execution for msec microseconds.
 */
static macro proc microsec_delay(msec)
{
  macro expr cycles = (PAL_ACTUAL_CLOCK_RATE * msec) / 1000000;
  unsigned (log2ceil(cycles)) count;

  count = 0;
  do
  {
    count++;
  }
  while(count != cycles - 1);
}
// ------------------------------------------------------------
```