Simulation Session

Serial Digital interfaces

# introduction

In this session you will write some C software to communicate with serial devices

At the end of this session, you should be able to do the following:

## intended learning outcomes

* Describe the function of a UART
* Describe the function of the hardware hand-shaking control lines and how this can result in blocking
* Contrast how polling methods with blocking – both in terms of complexity and efficacy
* To use the SPI interface to communicate with on-board peripherals

# Rs232 hardware

You may have used peripheral devices connected to a personal computer via a serial cable. This will probably conform to one of the RS232 standards. Typical examples include modems, electronic instrumentation and data terminals. RS232 is an asynchronous peer-to-peer communication standard. RS485 is similar, but can include multiple devices as it is a half-duplex serial bus.

**Key terms**

**Asynchronous** – no common clock is used

**Half-duplex** – a communication protocol that demands that data communication can occur in one direction at a time

**bus** – a set of data lines that are shared by multiple devices

Although such interfaces are now considered a legacy interfaces, there are still a lot of devices being manufactured using serial communication. USB has become to new standard for computer communication, and brings with it many advantages, at the expense of much more complexity.

## TasK 1 – RS232 Line Drivers

RS232 can be used to connect systems that do not share a common clock and unrelated to each other. An example would be two Personal Computers or two data terminals. They are connected with shielded cable. The digital signals are too small and lack sufficient power to drive over such distances at any decent speed, so a RS232 line driver device is needed. In this next task we take a brief look at these devices. This task uses the original bit-banging technique developed in the previous session.

|  |  |
| --- | --- |
| TASK |  |
| 1. Open the project in the folder TASK1 – RS232 Line Driver 2. Run the project to see what it does. Ensure you can see the schematic 3. Right click on the schematic – the bottom two items allow you to select which sheet to view    1. Take a close look at sheet 1 – you can see the serial data line connected to the RS232 line driver    2. TX stands for transmit    3. RX stands for receive    4. Now switch to sheet 2 4. Build and run the simulation again    1. The digital oscilloscope shows the actual voltage levels on the RS232 cable    2. The terminal shows the data received 5. Click on the one-shot button (on the oscilloscope) to capture some data | |
| TASK. Change the code to only send the character ‘H’. The data sent should resemble that shown in Figure 1.  Sketch and label a graph showing the actual voltage levels on the RS232 cable  Comment on the signal polarity and voltage  Note that each division on the instrument is 5V  TASK. Look closely at the function void sendRS232(const unsigned char c)  Can you draw a flow-chart / activity diagram (choose a suitable scheme) to represent the function of this code?  Which bit of which PORT is the serial data transmitted on? | |

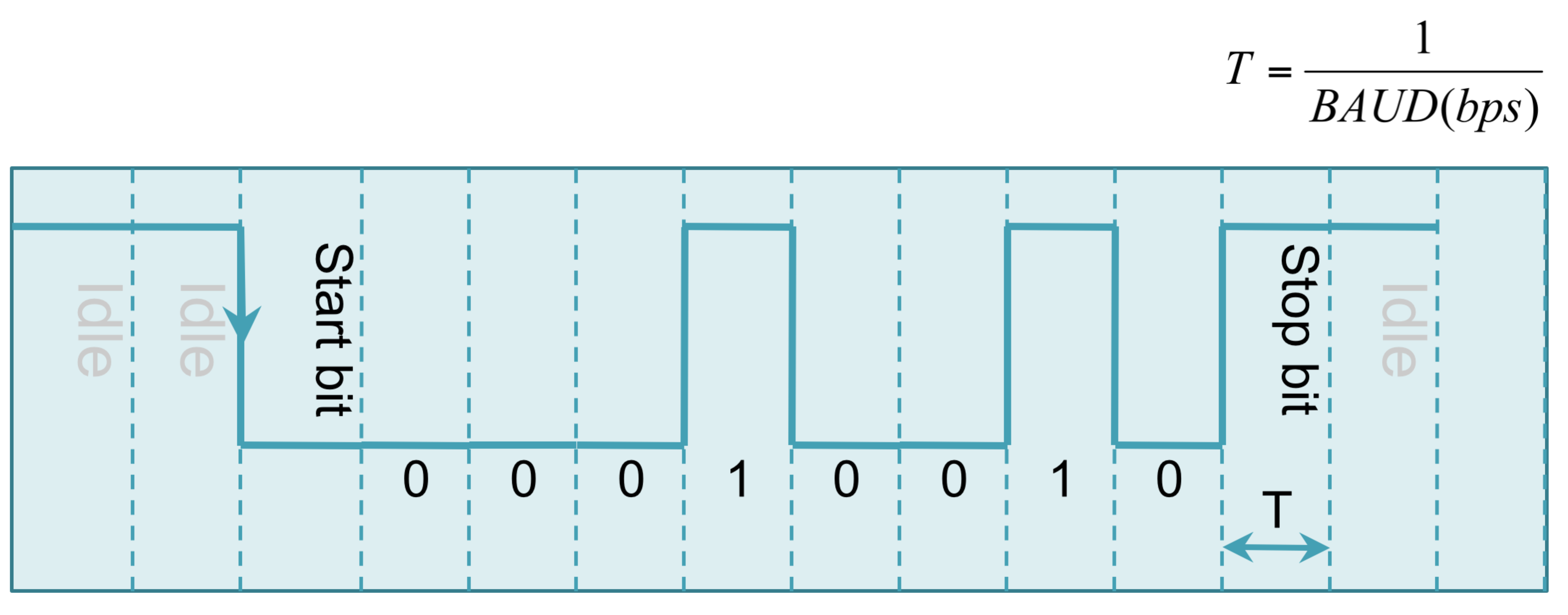


Figure 1 - RS232 bit pattern for the character 'H' (2400-8-N-1, no flow control)

**Notes**:

This example uses bit banging to generate a serial data protocol. Once the start bit is set, each bit is set after a fixed time interval. There is no clock link between the microcontroller and the terminal.

A very similar serial communication was once the basis the university computing network (before the internet)! Although considered legacy for personal computing, physical terminals still exist. Today you might use an application such as PuTTY (for MS Windows) as a terminal. In the embedded world, this is often useful for monitoring computers and troubleshooting (e.g. boot up sequences, before device drivers have been loaded). Some modern modules, such as GPRS, GPS, motion sensors, are packaged with serial communications. This makes them very simple to test using a computer terminal (such as PuTTY).

## TASK 2 – the uart

Bit-banging is usually only necessary if you are required to communicate with a non-standard protocol. RS232 and RS485 however are common protocols which are supported by a large number of devices, including the PIC24.

The more conventional and efficient mechanism for communication over RS232/485 is via a ‘Universal Asynchronous Receiver Transmitter’ or **UART**. The UART, once configured, performs the following roles for you:

* Converts outgoing parallel (7, 8 or 9 bit) data to serial (transmitter)
* Generates the required protocol bits (Start bit, stop bit(s), parity)
* Converts incoming serial data to parallel data (receiver)
* Locates and checks the required protocol bits
* Provides status information ( e.g. errors )
* Provides additional hardware control lines (RTS / CTS) – we will look at these
* Buffers data in both directions (4 bytes on the PIC24)
* Can often generate interrupts

In the simplest mode, you set up the UART and read and write data to it like any other register. However, if you want to ensure your data is received or that there is actually data to read, you need to do a little more work. This is where the CTS / RTS control lines come in.

1. Open the project file in the folder TASK2 - UART Chat
2. Build and run - Ensure the schematic is visible on the screen
   1. When the terminal U1 presents the message “Type Message Here”, click on terminal U1 and type a message. When you have finished, press return
   2. You should now see the message “Type Message Here” in the terminal U2. Click on terminal U2 and type a response
3. This is a very basic half-duplex communication system –
   1. it’s rather like the old radio systems where only one person can speak at a time
   2. one person speaks and then says ‘over’ (return key in our case) thus signaling the other user to respond.
4. Let’s now look through the code (on the next page)

**while** (1)

{

//Write a prompt on U1

putsU1\_blocking("Type Message Here >");

//Read response from keyboard UNTIL a return character is entered

**do** {

data\_rx = getU1\_blocking(); //Read from keyboard in term U1

putU2\_blocking(data\_rx); //Echo in U2

} **while** (data\_rx != '\r');

//Write a prompt on U2

putsU2\_blocking("Type Message Here >");

//Read response from keyboard UNTIL a return character is entered

**do** {

data\_rx = getU2\_blocking(); //Read from keyboard in term U2

putU1\_blocking(data\_rx); //Echo in U1

} **while** (data\_rx != '\r');

} //end while

The first prompt is written to the **terminal device** U1. A **terminal device** is one that receives characters from a computer and displays them on a screen. It also allows a user to type data and sends the characters back to the computer. This is all done over a serial connection.

//Write a prompt on U1

putsU1\_blocking("Type Message Here >");

Note this is a blocking function that sends a string of characters to a terminal.

1. Step **into** the code until you enter a function called “putU1\_blocking”. This is the code that writes a single character to the UART
2. Look at the sequence of events
   1. Wait for CTS (wait for the terminal device to indicate it is ready)

void putU1\_blocking( char c )

{

//Wait on a clear to send from the other end

**while** ( CTS1 == NOT\_RTS );

//Wait if transmit buffer is full

**while** ( U1STAbits.UTXBF);

//Write data to buffer (and transmit)

U1TXREG = c;

}

* 1. If the UART buffer is full, wait
  2. Send the data to the UART buffer (and hence transmit the data)

1. There are two instances of blocking here

|  |  |
| --- | --- |
| Question |  |
| If we did not use the RTS / CTS control lines, we would not bother to block on CTS. What do you think might happen in this case? | |
| Answer: | |

1. Now moving on. We’ve sent a string “Type Message Here >” to the terminal. We now want to wait for the user to type a message **and** echo it onto the second terminal U2. The code to do this is shown below

//Read response from keyboard UNTIL a return character is entered

**do** {

data\_rx = getU1\_blocking(); //Read from keyboard in term U1

putU2\_blocking(data\_rx); //Echo in U2

} **while** (data\_rx != '\r');

* 1. The getU1\_blocking() function **blocks** until there is a character available from the UART. If no data is available, then it simply waits and the CPU does nothing[[1]](#footnote-1).
  2. When a character becomes available, it is returned and stored in the variable data\_rx
  3. The character is written to the second terminal U2 using the putU2\_blocking function
  4. The **do-while loop** repeats the whole sequence **unless** the received character is the return key (‘\r’)

1. The remaining code repeats the above, only with the roles reversed – data is read from U2 and echoes to U1

This code illustrates an example of poor design. For one, it uses half-duplex communication which is not suited to human communication – you will have already experienced at least one form of superior **full-duplex** communication (human speech, telephone, Facebook messenger, SMS text messaging, etc…)

|  |  |
| --- | --- |
| Question |  |
| Fundamentally, why is this code limited to half-duplex communication? | |
| Answer: | |
| What alternative programming strategies could we use to get around the limitations shown here? | |
| Answer: | |

## task 3 – full duplex communicaton using polling

In this example we **begin** to overcome the limitations in the previous example. **Key to this is avoidance of blocking function calls**.

We have two fundamental options:

* **Polling** – reading / writing to hardware without waiting (as fast as possible)
* **Interrupts** – allowing the hardware (UART) to notify us of a change in state (character received; error; buffer status etc..)

1. Open the project file in the folder “TASK3 - UART Chat FD” (where FD stands for full-duplex)
2. This example uses polling to interact with the hardware. As you will see,
   1. It is far less simple than the previous example, but there is no blocking.
   2. It can process 2-way full-duplex communication
   3. It does not present the 2-way communication very cleanly, but that is a cosmetic issue which could be resolved later
3. Build and runthe code
   1. You can now type into either terminal in any order
   2. what you type will be echoed on both terminals

Now let’s look more closely at how we achieve this. The main loop is shown below. We will now walk through this code.

**while** (1) {

//Poll U1 for new data.

//If it has data (U1HasSentData==1), U1rx will be overwritten

**if** ( getU1\_polling(&U1rx) == 1 ) {

U1HasReceivedData = 1;

U2tx = U1rx;

}

//Poll U2 for new data

**if** ( getU2\_polling(&U2rx) == 1 ) {

U2HasReceivedData = 1;

U1tx = U2rx;

}

//If there is data received from U1, send it to U2

**if** (U1HasReceivedData) {

//Try and send the data to U2

U2HasSentData = putU2\_polling(U2tx);

//Has the data been sent yet?

**if** (U2HasSentData) {

//If yes, the transaction is complete

U1HasReceivedData = 0;

} //end if

} //end if

//If there is data received from U2, send it to U1

**if** (U2HasReceivedData) {

//Try and send the data to U2

U1HasSentData = putU1\_polling(U1tx);

//Has the data been sent yet?

**if** (U1HasSentData) {

//If yes, the transaction is complete

U2HasReceivedData = 0;

} //end if

} //end if

// DO WHAT EVER YOU LIKE IN THE REST OF THE TIME!!!

} //end while

### Reading data using polling

The function getU1\_polling checks if the UART has any data, and if so, returns a value of:

* 1-to indicate data was actually read;
* 0 – to indicate no data was available

//Poll U1 for new data.

//If it has data (U1HasSentData==1), U1rx will be overwritten

**if** ( getU1\_polling(&U1rx) == 1 ) {

U1HasReceivedData = 1;

U2tx = U1rx;

}

We call the function getU1\_polling (shown below), passing the **address** of the variable U1rx that is used to store any received data. Note that getU1\_polling is a non-blocking function.

Typical with any function that is interacting with hardware, getU1\_polling uses a finite state machine. Remember that:

* This function is called many times a second
* The state variable is ‘**static’** – its contents is preserved between calls
  + It does not always change state
* It returns data by overwriting rxData ONLY when there is data available
  + The function returns a 1 is data is available

#define RTS 0

#define NOT\_RTS 1

**unsigned** **int** getU1\_polling(char \*rxData)

{

**static** **unsigned** **int** state = 0; //First time this is called, state=0

unsigned **int** dataIsReceived = 0;

**switch** (state)

{

**case** 0:

RTS1 = RTS; //Tell the other end you are ready to receive data

state = 1;

break;

**case** 1:

**if** (U1STAbits.URXDA == 1) //If Receiver data has arrived...

{

\*rxData = U1RXREG; //read the buffer

dataIsReceived = 1; //Flag that data is now received

RTS1 = NOT\_RTS; //Pull RTS back high

state = 0; //Go back to state 0

} //end if

**break**;

**default**:

state = 0;

**break**;

} //end switch

**return** dataIsReceived;

}

This function has 2 states: The state diagram is shown below.

Data Received=0

Data Received=1/RTS=1

### Writing data using polling

Now we look at the equivalent function, putU1\_polling (shown below). This is the function used to write a byte to the UART. Again, it is non-blocking and has similar properties to the previous example:

* This function is called many times a second and is non-blocking
* The state variable is ‘**static’** – its contents is preserved between calls
  + It does not always change state
* It writes data to the UART ONLY when the UART is able to receive it
  + The function returns a 1 if data was sent

#define RTS 0

#define NOT\_RTS 1

unsigned int putU1\_polling(const char c)

{

static int state=0;

unsigned int sent = 0;

switch (state) {

case 0: //waiting for CTS

//CTS received

if (CTS1 == RTS) state = 1;

break;

case 1: //Wait for transmit buffer to become empty

//Ok, buffer is not full - proceed

if (U1STAbits.UTXBF == 0) state = 2;

break;

case 2:

//CTS received, buffer is not full - we can now send some data

U1TXREG = c; //Write data

sent = 1; //Flag that data has been sent

state = 0; //Return to state 0

break;

default: //should not occur, but just in case!

state=0;

break;

} //end switch

return sent;

}

|  |  |
| --- | --- |
| Question |  |
| How many states does this function use? | |
| Sketch and label a Finite State Diagram for this function | |
|  | |
| What is the purpose of the “default” state in the code? | |
|  | |

# ADVANCED - Configuration of the UART

The UART allows us to write data in parallel binary format without the need for bit banging. The code used to configure the UART is shown below:

TRISDbits.TRISD15 = 0; //Ready To Send - (RTS) is an output

TRISDbits.TRISD14 = 1; //Clear To Send (CTS) is an input

U1MODEbits.BRGH = 1; //BRGH = 1, so use a 1:4 divider

U1BRG = BAUD; //Set baud rate to 2400 for (assuming BRGH = 1)

U1MODEbits.UARTEN = 1; //Enable the UART

U1STA = 0x0400; //Enable Transmit and reset all other bits

RTS1 = NOT\_RTS; //Not ready, default mode (active low)

The UART data output and input pins are fixed. If we choose to use CTS/RTS, then pins have to be assigned. Here D15 and D14 are used.

The UART has its own timer to enable the data to be written at the correct rate – the rate is known as the BAUD rate. Here we have used a very slow rate of 2400. The BAUD is set using two registers, BRGH and UxBRG, and the following formula:



Or conversely,



where BRHG is a single bit, which selects the clock divider (1:4 or 1:16). We will use 1:4 to get more precision; UxBRG is essentially a 16-bit period register – we calculate this to set the Baud rate.

**For example**: Calculate the U1BRG for a Baud rate of 2400

Set BRGH=1 (1:4 pre-scale)

We are running at 32MHz (16 MIPS)



We now have two choices, U1BRG-1665 or 1666, so which is better?

Plug both values back into the first formula:



The choice of 1666 results in the closest BAUD rate.

|  |  |
| --- | --- |
| TASK |  |
| Configure both UARTs to run at 9600 baud. Show your calculations and modify the code accordingly.  ( You will also need to change the properties of each terminal in the schematic to match it - stop the simulation, click the console twice in succession, edit the properties ) | |

1. With the exception of interrupt service routines [↑](#footnote-ref-1)