Lab 4: Using UART

COEN-4720 Embedded Systems

Cristinel Ababei
Dept. of Electrical and Computer Engineering, Marquette University

1. Objective

The objective of this lab is to utilize the Universal Asynchronous Receiver/Transmitter (UART) to connect the LandTiger 2.0 board to the host computer. We'll continue to discuss the concept of interrupts. In the example project, we send characters to the microcontroller unit (MCU) of the board by pressing keys on the keyboard. These characters are sent back (i.e., echoed, looped-back) to the host computer by the MCU and are printed in a terminal window.

2. UART

The most basic method for communication with an embedded processor is asynchronous serial. It is implemented over a symmetric pair of wires connecting two devices (referred as host and target here, though these terms are arbitrary). Whenever the host has data to send to the target, it does so by sending an encoded bit stream over its transmit (TX) wire. This data is received by the target over its receive (RX) wire. The communication is similar in the opposite direction. This simple arrangement is illustrated in Fig.1. This mode of communications is called "asynchronous" because the host and target share no time reference (no clock signal). Instead, temporal properties are encoded in the bit stream by the transmitter and must be decoded by the receiver.

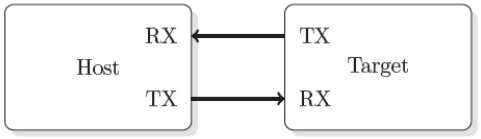

Figure 1 Basic serial communication.

A commonly used device for encoding and decoding such asynchronous bit streams is a Universal Asynchronous Receiver/Transmitter (UART). UART is a circuit that sends parallel data through a serial line. UARTs are frequently used in conjunction with the RS-232 standard (or specification), which specifies the electrical, mechanical, functional, and procedural characteristics of two data communication equipment.

A UART includes a transmitter and a receiver. The transmitter is essentially a special shift register that loads data in parallel and then shifts it out bit by bit at a specific rate. The receiver, on the other hand, shifts in data bit by bit and reassembles the data. The serial line is '1' when it is idle. The transmission starts with a start-bit, which is '0', followed by data-bits and an optional parity-bit, and ends with stop-bits, which are '1'. The number of data-bits can be 6, 7, or 8. The optional parity bit is used for error detection. For odd parity, it is set to '0' when the data bits have an odd number of '1's. For even parity, it is set to '0' when the data-bits have an even number of '1's. The number of stop-bits can be 1, 1.5, or 2. The transmission with 8 data-bits, no parity, and 1 stop-bit is shown in Fig.2 (note that the LSB of the data word is transmitted first).

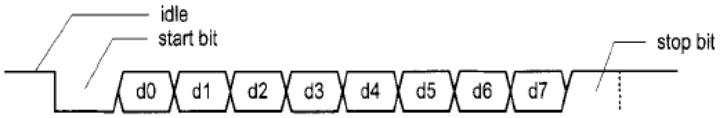

Figure 2 Transmission of a byte.

No clock information is conveyed through the serial line. Before the transmission starts, the transmitter and receiver must agree on a set of parameters in advance, which include the baud-rate (i.e., number of bits per second), the number of data bits and stop bits, and use of parity bit.

To understand how the UART's receiver extracts encoded data, assume it has a clock running at a multiple of the baud rate (e.g., 16x). Starting in the idle state (as shown in Fig.3), the receiver "samples" its RX signal until it detects a high-low transition. Then, it waits 1.5 bit periods (24 clock periods) to sample its RX signal at what it estimates to be the center of data bit 0. The receiver then samples RX at bit-period intervals (16 clock periods) until it has read the remaining 7 data bits and the stop bit. From that point this process is repeated. Successful extraction of the data from a frame requires that, over 10.5 bit periods, the drift of the receiver clock relative to the transmitter clock be less than 0.5 periods in order to correctly detect the stop bit.

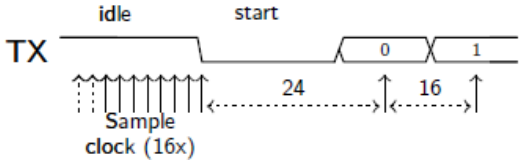

Figure 3 Illustration of signal decoding.

UARTs can be used to interface to a wide variety of other peripherals. For example, widely available GSM/GPRS cell phone modems and Bluetooth modems can be interfaced to a microcontroller UART. Similarly GPS receivers frequently support UART interfaces.

The NXP LPC1768 microcontroller includes four such devices/peripherals called UARTs: UART0/2/3 and UART1. See pages 307 and 327 of the LPC17xx User's Manual [1]. See also page 27 of the Datasheet [2]. UART0 and UART2 of the microcontroller are connected on the LandTiger 2.0 board to the SP3232 (U10 on the board's schematic diagram), which converts the logic signals to RS-232 voltage levels. This connection is realized from pins {*P0.2 and P0.3*} and { *P0.10 and P0.11*} of the microcontroller to the pins {*9, 10, 11, and 12*} SP3232 chip. To confirm this, please take a look at the board's schematic diagram, provided as file **0_HY_LandTiger_SCH.pdf** (page 6) inside the files archive of lab#2. Also, take some time to read Section 2.12 of the board's manual, provided as file **1_HY_LandTiger_v20_user_manual_v11.pdf** (pages 18-19) inside the files archive of lab#2.
The SP3232 chip drives the two COM1 and COM2 represented by the two *female* DB9 connectors on the board.

For more details on UART and RS-232, please read references [1-4].

In this lab we will explore serial communication between the (target) LPC1768 UART and a serial communication port of the host PC.

3. Example1: Microcontroller "echoes" back the characters sent by host computer

(a) Experiment

In the first example of this lab, we'll use an adapted (to the LandTiger 2.0 board) example project that comes with the "LPC1700 code bundle" [5]. The LPC1700 Code Bundle is a free software package from NXP that demonstrates the use of the built-in peripherals on the NXP LPC17xx series of microcontrollers. The example software includes a common library, peripheral APIs, and test modules for the APIs. These examples were put together for ARM's boards. However, because LandTiger 2.0 board bears a lot of similarities with ARM's MCB1700 board, most of these examples can be adapted fairly easily to make them work on the LandTiger 2.0 board. For instance, in this lab we work with an adapted version of the UART/ example from the "LPC1700 code bundle".

All the required source files for this example are located under **Example1/** folder that comes with the archive of files provided for this lab. Create a new uVision project and add all the provided files to your project. **Before doing anything, first make sure you remove the ISP and RST jumpers of the LandTiger 2.0 board**; these are the jumpers marked as JP6 and JP7 on the board, next to the actual COM connectors.

Step 1: Connect the board (COM1) and the host computer using the serial cable.

Step 2: Familiarize yourself with the following files:
--**uart.c**: contains the UART0 / UART2 handlers/driver functions
--**uarttest.c**: contains a small test program

Step 3: Build the project and download to target.

Step 4: Establish a Terminal connection
--Method 1: If your Windows is XP or older, you can use HyperTerminal:
--Start HyperTerminal by clicking Start - > All Programs -> Accessories -> Communications -> HyperTerminal
--Connect HyperTerminal to the serial port that is connected to the COM0 port of the evaluation board for example. For the HyperTerminal settings you should use: COM1 (double check that it the host's serial port is indeed COM1in your case – you can do that by Start->Control Panel->System->Hardware->Device Manager and click on Ports (COM & LPT); if in your case it's a different port number then use that; for example, in my case as I use the USB to serial adapter with my laptop, the port is COM14), baud rate 9600, 8 data bits, no parity, 1 stop bit, and no flow control.
*--**Method 2:** If your Windows is 7 or newer, you first must make sure you download and/or install a serial connection interface (because HyperTerminal is not part of Windows 7 or later). On the machines in the lab, you can use Putty (http://www.putty.org):*
--Start->All Programs->Putty->Putty
--Then, select Connection, Serial and type COM1 (or whatever is in your case; find what it is as described in Method 1), baud rate 9600, 8 data bits, no parity, 1 stop bit, and no flow control.
--Click on Session and choose the type of connection as Serial. Save the session for example as "lab4".
--Finally, click on Open; you should get HyperTerminal like window.
*--**Method 3:** You can use other programs such as:*

TeraTerm (http://logmett.com/index.php?/products/teraterm.html) or
HyperSerialPort (http://www.hyperserialport.com/index.html) or
RealTerm (http://realterm.sourceforge.net) or
CoolTerm (http://freeware.the-meiers.org), etc.

Step 5: Type some text. The text should appear in the HyperTerminal window. This is the result of: First, what you type is sent to the microcontroller via the serial port (that uses a UART). Second, the MCU receives that text via its own UART and echoes (sends back) the typed characters using the same UART. The host PC receives the echoed characters that are displayed in the HyperTerminal.
Disconnect the serial cable from COM1 and connect it to COM2 port on the LandTiger 2.0 board. The behavior of project should be the same, as I hope you expected.

(b) Brief program description

Looking at the **main()** function inside the **uarttest.c** file we can see the following main actions:
--UART0 and UART2 are initialized:

```
UARTInit(0, 9600);
UARTInit(2, 9600);
```

--A while loop which executes indefinitely either for UART0 or for UART2. The instructions executed inside this loop (let's assume the UART0) are:

- Disable **Receiver Buffer Register (RBR)**, which contains the next received character to be read. This is achieved by setting all bits of the **Interrupt Enable Register (IER)** to '0' except bits THRE and RLS. In this way the LSB (i.e., bit index 0 out of 32 bits) of IER register is set to '0'. The role of this bit (as explained in the LPC17xx User's Manual [1], page 311) when set to '0' is to disable the **Receive Data Available** interrupt for UART0.
- Send back to the host the characters from the buffer UART0_Buffer. This is done only if there are characters in the buffer (the total buffer size is 40 characters) which have been received so far from the host PC. The number of characters received so far and not yet echoed back is stored and updated in UART0_Count.
- Once the transmission of all characters from the buffer is done, reset the counter UART0_Count.
- Enable **Receiver Buffer Register (RBR)**. This is achieved by setting to '1' the LSB of IER, which in turn is achieved using the mask IER_RBR.

(c) Source code discussion

Please take your time now to thoroughly browse the source code in files **uart.c** and **uarttest.c** files. Open and read other files as necessary to track the declarations and descriptions of variables and functions.

For example, in **uarttest.c** we see:
--A function call **SystemInit();**
This is **described** in source file **system_LPC17xx.c**. Place the mouse cursor on SystemInit(), then right click and select Go to the Definition of 'SystemInit'. Try to understand how this function is implemented; what each of the source code lines do?
--A function call **UARTInit(0, 9600);**
This function is described in source file **uart.c**. The declaration of this function is done inside the header file **uart.h**. Open these files and read the whole code; try to understand each line of the code.

--An instruction:

```
LPC_UART0->IER = IER_THRE | IER_RLS; // Disable RBR
```

Based on what's been studied in the previous labs we should know already that **LPC_UART0** is an address of a memory location – the UART0 peripheral is "mapped" to this memory location. Starting with this memory location, several consecutive memory locations represent "registers" associated with this peripheral/device called UART0. All 14 such registers (or memory locations) are listed on page 309 of the User Manual. Utilizing this peripheral is done through reading/writing into these registers according to their meaning and rules described in the manual. For example, one of the 14 registers associated with UART0 is **Interrupt Enable Register (IER)**.

From a C programming perspective, **LPC_UART0** is declared inside header file **LPC17xx.h**:

```
#define LPC_UART0             ((LPC_UART_TypeDef      *) LPC_UART0_BASE    )
```

Which also declares what **LPC_UART0_BASE** as:

```
#define LPC_UART0_BASE        (LPC_APB0_BASE + 0x0C000)
```

Where **LPC_APB0_BASE** is declared at its turn in the same file:

```
#define LPC_APB0_BASE         (0x40000000UL)
```

This effectively makes **LPC_UART0_BASE** to have value: **0x4000C000**, which not surprisingly coincides with what is reported on page 14 of the LPC17xx User's Manual!

Furthermore, the **Interrupt Enable Register (IER)** contains individual interrupt enable bits for the 7 potential UART interrupts. The IER register for UART0 is "mapped" (or associated with) to memory address **0x4000C004** as seen on page 309 of the LPC17xx User's Manual. This fact is captured in the **struct** declaration of **LPC_UART_TypeDef** inside the header file **LPC17xx.h** (open this file and double check it!). As a result, in our C programming, we can refer to the IER register as in the instruction that we are currently discussing: **LPC_UART0->IER**, which basically stores/represents the address **0x4000C004**.

In addition, note that **IER_THRE** and **IER_RLS** are declared inside the header file **uart.h** as:

```
#define IER_THRE  0x02
#define IER_RLS   0x04
```

Which are utilized as masks in our instruction:

```
LPC_UART0->IER = IER_THRE | IER_RLS; // Disable RBR
```

So, finally as we see, the effect of this instruction is simply to turn '1' bit index 1 (the second LSB out of 32 bits) and bit index 2 (the third LSB out of 32 bits) of the IER register! All other bits are set to '0'.

Having bit index 1 of this register set to '1' enables the **Transmit Holding Register Empty (THRE)** flag for UART0 – see Table 276, page 311 of the LPC17xx User's Manual. Having bit index 2 of this register set to '1' enables the UART0 RX line status interrupts. As already said, all other bits are set therefore via this masking to '0'. This includes the LSB (i.e., bit index 0 out of 32 bits) of IER register, which is set to '0'. The role of this bit (as explained in the LPC17xx User's Manual on page 311) when set to '0' is to disable the **Receive Data Available** interrupt for UART0.

You should be able now to explain what the following instruction does:

```
LPC_UART0->IER = IER_THRE | IER_RLS | IER_RBR; // Re-enable RBR
```

Summarizing, what the code inside **uarttest.c** does is:

--disable receiving data

--send back data to the PC from the buffer (i.e., array variable UART0_Buffer)

--reset counter of characters stored in buffer

--enable receiving data

Note: As mentioned in previous labs, in general, one would not need to worry about these details about addresses to which registers are mapped. It would be sufficient to just know of, for example, the definition

*and declaration of **LPC_UART_TypeDef** inside the header file **LPC17xx.h**. To the C programmer, it is transparent to what exact address the register IER is mapped to for example. However, now at the beginning of this class, it's instructive to track these things so that we get a better global picture of these concepts. It also forces us to get better used with the LPC17xx User's Manual and the datasheets.*

Notice that inside the source file **uart.c** we have these two function descriptions:

```
void UART0_IRQHandler (void) {...}
void UART2_IRQHandler (void) {...}
```

which are not called for example inside **uarttest.c**, but they appear inside **startup_LPC17xx.s**:

```
DCD     UART0_IRQHandler
DCD     UART2_IRQHandler
```

The function **void UART0_IRQHandler (void)** is the UART0 interrupt handler. Its name is **UART0_IRQHandler** because it is named like that by the startup code inside the file **startup_LPC17xx.s**. **DCD** is an assembler directive (or pseudo-op) that defines a 32-bit constant.
Inside this function, we see a first instruction:

```
IIRValue = LPC_UART0->IIR;
```

What does it do? It simply reads the value of **LPC_UART0->IIR** and assigns it to a variable whose name is IIRValue. **LPC_UART0->IIR** is the value of the register **IIR** (Interrupt IDentification Register - identifies which interrupt(s) are pending), which is one of several (14 of them) registers associated with the UART0 peripheral/device. You can see it as well as the other registers on page 309 of the User Manual. Take a while and read them all. The fields of the interrupt register **IIR** are later described on page 312 in the User Manual. Take another while and read them all on page 312.
Next, inside **uart.c** we see:

```
  IIRValue >>= 1;            /* skip pending bit in IIR */
  IIRValue &= 0x07;          /* check bit 1~3, interrupt identification */
```

Which shifts right with one bit IIRValue and then AND's it with 0x07. This effectively "zooms-in" onto the field formed by bits index 1-3 from the original **LPC_UART0->IIR**, bits which are now in the position bits index 0-2 of IIRValue variable.
Going on, we find an "if" instruction with several branches:

```
  if ( IIRValue == IIR_RLS )        /* Receive Line Status */
  {...
  }
  else if ( IIRValue == IIR_RDA )   /* Receive Data Available */
  {...
  }
  else if ( IIRValue == IIR_CTI )   /* Character timeout indicator */
  {...
  }
  else if ( IIRValue == IIR_THRE )  /* THRE, transmit holding register empty */
  {...
  }
```

Read Table 277, page 312 in the User Manual to see the meaning of the three bits 1-3 from the original IIR register:

```
011 1 - Receive Line Status (RLS).
010 2a - Receive Data Available (RDA).
110 2b - Character Time-out Indicator (CTI).
001 3 - THRE Interrupt
```

For each of these situations, something else is done inside the corresponding branch of the "if" instruction above. In other words, we first identify the interrupt, and for each ID we do something else. If none of the expected IDs is found, we do nothing. **Please take your time now to explain what's done in each of these cases**. Read pages 312 in the User Manual for this. This is very important in order to understand the overall operation of the example of this lab.

4. Lab Assignment

1) Describe in less than a page (typed, font size 11, single line spacing) all instructions inside the function **void UART0_IRQHandler (void)** for each of the branches of the main "if" instruction. Include this in your lab report.
2) Modify Example 1 such that the characters typed on the host's keyboard are also displayed on the LCD display on the board.

5. Credits and references

[1] LPC17xx user's manual; http://www.nxp.com/documents/user_manual/UM10360.pdf (part of lab#2 archive of files)
[2] NXP LPC17xx Datasheet;
http://www.nxp.com/documents/data_sheet/LPC1769_68_67_66_65_64_63.pdf (also part of lab#2 files archive of files)
[3]
--Jonathan W. Valvano, Embedded Systems: Introduction to Arm Cortex-M3 Microcontrollers, 2012. (Chapters 8,9)
--Pong P. Chu, FPGA Prototyping by VHDL Examples: Xilinx Spartan-3 Version, Wiley 2008. (Chapter 7)
--Lab manual of course http://www.cs.indiana.edu/~geobrown/book.pdf (Chapter 5)
--EE 472 Course Note Pack, Univ. of Washington, 2009,
http://abstract.cs.washington.edu/~shwetak/classes/ee472/notes/472_note_pack.pdf (Chapter 8)
[4] UART entry on Wikipedia (click also on the references therein for RS-232);
http://en.wikipedia.org/wiki/Universal_asynchronous_receiver/transmitter
[5] LPC1700 Code Bundle;
Download: http://ics.nxp.com/support/software/code.bundle.lpc17xx.keil/
Documentation:
http://ics.nxp.com/literature/presentations/microcontrollers/pdf/code.bundle.lpc17xx.keil.uart.pdf
[6] Pluggable USB to RS-232 DB9 Serial Adapter;
Amazon: http://www.amazon.com/Plugable-Adapter-Prolific-PL2303HX-Chipset/dp/B00425S1H8/ref=sr_1_1?ie=UTF8&qid=1359988639&sr=8-1&keywords=plugable+usb+to+serial
Tigerdirect: http://www.tigerdirect.com/applications/SearchTools/item-details.asp?EdpNo=3753055&CatId=464