**EE118IU**

**EMBEDDED SYSTEM LABORATORY**

**LAB 4 :**

**UART & ADC & GPIO**

**WITH MULTI-TASK PROGRAMMING**

**Student:**

**Student ID:**

**Class:**

**Date:**

I. LAB OBJECTIVES

Upon completion of this lab, you will able to:

* Communicate LPC1768 with PC through UART protocol and handle interrupts and errors occurring during operation.
* Be able to use ADC module in LPC1768 together with UART and GPIO module using multitask programming.
* Student have ability to develop multi-task programming

**II. PRE-LAB :**

* Simple steps in in setting up peripherals for microcontroller are:

- Setting “Control registers”, especially PCONP (Power Control for Peripherals) register.

- Setting “Pin Connect Block registers”, especially PINSEL (Pin Function Select) register  
(from PINSEL0 to PINSEL10).

- Setting corresponding registers to peripherals we are going to use

* PCONP (Power Control for Peripherals) register and Pin Connect Block registers are presented in details in “Lab 1 – Tutorial”.
* In following sections, registers used for UART and ADC will be mentioned in detail.
  1. UART Registers:
     1. Overview:
* Three following tables are extracted from Table 271, page 310 in “User Manual”.
* Registers used to setup the UART.

|  |  |
| --- | --- |
| **Generic Name** | **Description** |
| DLL | Divisor Latch LSB. Least significant byte of the baud rate divisor value. The full divisor is used to generate a baud rate from the fractional rate divider. |
| DLM | Divisor Latch MSB. Most significant byte of the baud rate divisor value. The full divisor is used to generate a baud rate from the fractional rate divider. |
| FDR | Fractional Divider Register. Generates a clock input for the baud rate divider. |
| IER | Interrupt Enable Register. Contains individual interrupt enable bits for the 7 potential UART interrupts. |
| FCR | FIFO Control Register. Controls UART FIFO usage and modes. |
| LCR | Line Control Register. Contains controls for frame formatting and break generation |

* Registers used to handle UART Interrupts.

|  |  |
| --- | --- |
| **Generic Name** | **Description** |
| IRR | Interrupt ID Register. Identifies which interrupt(s) are pending. |
| LSR | Line Status Register. Contains flags for transmit and receive status, including line errors. |

* Registers used to send and receive data from UART.

|  |  |
| --- | --- |
| **Generic Name** | **Description** |
| RBR | Receiver Buffer Register. Contains the next received character to be read. |
| THR | Transmit Holding Register. The next character to be transmitted is written here |

* Other registers that we don’t use yet: SCR, ACR, ICR, TER.
* In Keil uVision, to access “UART0 Registers”, use:

**LPC\_UART0;**

For example, to set value of “FCR register” of UART0, use:

**LPC\_UART0->FCR = 0x07;**

* Baud rate calculation (from function (4), page 323 in “User Manual”):
  + 1. DLL and DLM Registers:

DLL (UARTn Divisor Latch LSB) register bit description (From table 274, page 312 in “User Manual”).

|  |  |  |
| --- | --- | --- |
| **Bit** | **Symbol** | **Description** |
| 7:0 | DLLSB | The UARTn Divisor Latch LSB Register, along with the UnDLM register, determines the baud rate of the UARTn. |
| 31:8 | - | Reserved |

DLM (UARTn Divisor Latch MSB) register bit description (From table 274, page 312 in “User Manual”).

|  |  |  |
| --- | --- | --- |
| **Bit** | **Symbol** | **Description** |
| 7:0 | DLLSB | The UARTn Divisor Latch MSB Register, along with the UnDLL register, determines the baud rate of the UARTn. |
| 31:8 | - | Reserved |

* + 1. FDR register:

FDR (UARTn Fractional Divider) register bit description (From table 286, page 322 in “User Manual”).

|  |  |  |
| --- | --- | --- |
| **Bit** | **Symbol** | **Description** |
| 3:0 | DIVADDVAL | Baud-rate generation pre-scaler divisor value. If this field is 0, fractional baud-rate generator will not impact the UARTn baudrate. |
| 7:4 | MULVAL | Baud-rate pre-scaler multiplier value. This field must be greater or equal 1 for UARTn to operate properly, regardless of whether the fractional baud-rate generator is used or not. |
| 31:8 |  | Reserved |

* + 1. IER register:

IER (UARTn Interrupt Enable) register bit description (From table 276, page 312 in “User Manual”).

|  |  |  |
| --- | --- | --- |
| **Bit** | **Symbol** | **Description** |
| 0 | RBR Interrupt Enable | Enables the Receive Data Available interrupt for UARTn. It also controls the Character Receive Time-out interrupt. |
| 1 | THRE Interrupt Enable | Enables the THRE interrupt for UARTn. The status of this can be read from UnLSR[5]. |
| 2 | RX Line Status Interrupt Enable | Enables the UARTn RX line status interrupts. The status of this interrupt can be read from UnLSR[4:1]. |
| 7:3 | - | Reserved |
| 8 | ABEOIntEn | Enables the end of auto-baud interrupt. |
| 9 | ABTOIntEn | Enables the auto-baud time-out interrupt. |
| 31:10 | - | Reserved |

* + 1. FCR register:

FCR (UARTn FIFO Control) register bit description (From table 279, page 315 in “User Manual”).

|  |  |  |
| --- | --- | --- |
| **Bit** | **Symbol** | **Description** |
| 0 | FIFO Enable | 0: UARTn FIFOs are disabled. Must not be used in the application.  1: Active high enable for both UARTn Rx and TX FIFOs and UnFCR[7:1] access. This bit must be set for proper UART operation. Any transition on this bit will automatically clear the related UART FIFOs |
| 1 | RX FIFO Reset | 0: No impact on either of UARTn FIFOs.  1: Writing a logic 1 to UnFCR[1] will clear all bytes in UARTn Rx FIFO, reset the pointer logic. This bit is self-clearing. |
| 2 | TX FIFO Reset | 0: No impact on either of UARTn FIFOs.  1: Writing a logic 1 to UnFCR[2] will clear all bytes in UARTn TX FIFO, reset the pointer logic. This bit is self-clearing. |
| 3 | DMA Mode Select | When the FIFO enable bit (bit 0 of this register) is set, this bit selects the DMA mode. |
| 5:4 | - | Reserved |
| 7:6 | RX Trigger Level | These two bits determine how many receiver UARTn FIFO characters must be written before an interrupt or DMA request is activated.  00: 1 character.  01: 4 characters.  10: 8 characters.  11: 14 characters. |
| 31:8 | - | Reserved |

* + 1. LCR register:

LCR (UARTn Line Control) register bit description (From table 280, page 316 in “User Manual”).

|  |  |  |
| --- | --- | --- |
| **Bit** | **Symbol** | **Description** |
| 1:0 | Word Length Select | 00: 5-bit character length  01: 6-bit character length  10: 7-bit character length  11: 8-bit character length |
| 2 | Stop Bit Select | 0: 1 stop bit.  1: 2 stop bit (1.5 if UnLCR[1:0]=00) |
| 3 | Parity Enable | 0: Disable parity generation and checking.  1: Enable parity generation and checking. |
| 5:4 | Parity Select | 00: Odd parity. Number of 1s in the transmitted character and the attached parity bit will be odd.  01: Even Parity. Number of 1s in the transmitted character and the attached parity bit will be even.  10: Forced "1" stick parity.  11: Forced "0" stick parity. |
| 6 | Break Control | 0: Disable break transmission.  1: Enable break transmission. Output pin UARTn TXD is forced to logic 0 when UnLCR[6] is active high. |
| 7 | Divisor Latch Access Bit | 0: Disable access to Divisor Latches.  1: Enable access to Divisor Latches. |
| 31:8 | - | Reserved |

* + 1. IRR register:

IRR (UARTn Interrupt Identification) register bit description (From table 277, page 313 in “User Manual”).

|  |  |  |
| --- | --- | --- |
| **Bit** | **Symbol** | **Description** |
| 0 | IntStatus | Interrupt status. Note that UnIIR[0] is active low. The pending interrupt can be determined by evaluating UnIIR[3:1].  0: At least one interrupt is pending.  1: No interrupt is pending. |
| 3:1 | IntId | Interrupt identification. UnIER[3:1] identifies an interrupt corresponding to the UARTn Rx or TX FIFO. All other combinations of UnIER[3:1] not listed below are reserved.  011: Receive Line Status (RLS).  010: Receive Data Available (RDA).  110: Character Time-out Indicator (CTI).  001: Transmitter Holding Register Empty (THRE). |
| 5:4 | - | Reserved |
| 7:6 | FIFO Enable | Copies of UnFCR[0]. |
| 8 | ABEOInt | End of auto-baud interrupt. True if auto-baud has finished successfully and interrupt is enabled. |
| 9 | ABTOInt | Auto-baud time-out interrupt. True if auto-baud has timed out and interrupt is enabled. |
| 31:10 | - | Reserved |

* + 1. LSR register:

LSR (UARTn Line Status) register bit description (From table 281, page 317 in “User Manual”).

|  |  |  |
| --- | --- | --- |
| **Bit** | **Symbol** | **Description** |
| 0 | Receiver Data Ready (RDR) | UnLSR0 is set when the UnRBR holds an unread character and is cleared when the UARTn RBR FIFO is empty. |
| 1 | Overrun Error (OE) | The overrun error condition is set as soon as it occurs. An UnLSR read clears UnLSR1. UnLSR1 is set when UARTn RSR has a new character assembled and the UARTn RBR FIFO is full. In this case, the UARTn RBR FIFO will not be overwritten and the character in the UARTn RSR will be lost. |
| 2 | Parity Error (PE) | When the parity bit of a received character is in the wrong state, a parity error occurs. An UnLSR read clears UnLSR[2]. Time of parity error detection is dependent on UnFCR[0] |
| 3 | Framing Error (FE) | When the stop bit of a received character is a logic 0, a framing error occurs. An UnLSR read clears UnLSR[3]. The time of the framing error detection is dependent on UnFCR0. Upon detection of a framing error, the Rx will attempt to resynchronize to the data and assume that the bad stop bit is actually an early start bit. However, it cannot be assumed that the next received byte will be correct even if there is no Framing Error. |
| 4 | Break Interrupt (BI) | When RXDn is held in the spacing state (all zeroes) for one full character transmission (start, data, parity, stop), a break interrupt occurs. Once the break condition has been detected, the receiver goes idle until RXDn goes to marking state (all ones). An UnLSR read clears this status bit. The time of break detection is dependent on UnFCR[0]. |
| 5 | Transmitter Holding Register Empty (THRE) | THRE is set immediately upon detection of an empty UARTn THR and is cleared on a UnTHR write. |
| 6 | Transmitter Empty (TEMT) | TEMT is set when both UnTHR and UnTSR are empty; TEMT is cleared when either the UnTSR or the UnTHR contain valid data. |
| 7 | Error in RX FIFO (RXFE) | UnLSR[7] is set when a character with a Rx error such as framing error, parity error or break interrupt, is loaded into the UnRBR. This bit is cleared when the UnLSR register is read and there are no subsequent errors in the UARTn FIFO. |
| 31:8 | - | Reserved |

* + 1. RBR register:

RBR (UARTn Receiver Buffer) register bit description (From table 272, page 311 in “User Manual”).

|  |  |  |
| --- | --- | --- |
| **Bit** | **Symbol** | **Description** |
| 7:0 | RBR | The UARTn Receiver Buffer Register contains the oldest received byte in the UARTn Rx FIFO. |
| 31:8 | - | Reserved |

* + 1. THR register:

THR (UARTn Transmit Holding) register bit description (From table 273, page 311 in “User Manual”).

|  |  |  |
| --- | --- | --- |
| **Bit** | **Symbol** | **Description** |
| 7:0 | THR | Writing to the UARTn Transmit Holding Register causes the data to be stored in the UARTn transmit FIFO. The byte will be sent when it reaches the bottom of the FIFO and the transmitter is available. |
| 31:8 | - | Reserved |

* 1. ADC Registers:
* Three following tables are extracted from Table 531, page 587 in “User Manual”.
* Registers used to setup the ADC.

|  |  |
| --- | --- |
| **Generic Name** | **Description** |
| ADCR | A/D Control Register. The ADCR register must be written to select the operating mode before A/D conversion can occur. |
| ADINTEN | A/D Interrupt Enable Register. This register contains enable bits that allow the DONE flag of each A/D channel to be included or excluded from contributing to the generation of an A/D interrupt. |

* Registers used to handle ADC Interrupts.

|  |  |
| --- | --- |
| **Generic Name** | **Description** |
| ADSTAT | A/D Status Register. This register contains DONE and OVERRUN flags for all of the A/D channels, as well as the A/D interrupt/DMA flag |

* Registers used to send and receive data from ADC.

|  |  |
| --- | --- |
| **Generic Name** | **Description** |
| ADGDR | A/D Status Register. This register contains DONE and OVERRUN flags for all of the A/D channels, as well as the A/D interrupt/DMA flag |
| ADDR0 to  ADDR7 | A/D Channel 0🡪7 Data Register. This register contains the result of the most recent conversion completed on channel 0🡪7. |

* + 1. ADCR register:

ADCR (A/D Control) register bit description (From Table 532, page 588 in “User Manual”).

|  |  |  |
| --- | --- | --- |
| **Bit** | **Symbol** | **Description** |
| 7:0 | SEL | Selects which of the AD0.7:0 pins is (are) to be sampled and converted. For AD0, bit 0 selects Pin AD0.0, and bit 7 selects pin AD0.7. In software-controlled mode, only one of these bits should be 1. In hardware scan mode, any value containing 1 to 8 ones is allowed. All zeroes is equivalent to 0x01. |
| 15:8 | CLKDIV | The APB clock (PCLK\_ADC0) is divided by (this value plus one) to produce the clock for the A/D converter, which should be less than or equal to 13 MHz. Typically, software should program the smallest value in this field that yields a clock of 13 MHz or slightly less, but in certain cases (such as a high-impedance analog source) a slower clock may be desirable. |
| 16 | BURST | 1: The AD converter does repeated conversions at up to 200 kHz, scanning (if necessary) through the pins selected by bits set to ones in the SEL field. The first conversion after the start corresponds to the least-significant 1 in the SEL field, then higher numbered 1-bits (pins) if applicable. Repeated conversions can be terminated by clearing this bit, but the conversion that’s in progress when this bit is cleared will be completed.  Remark: START bits must be 000 when BURST = 1 or conversions will not start. If BURST is set to 1, the ADGINTEN bit in the AD0INTEN register (Table 534) must be set to 0.  0: Conversions are software controlled and require 65 clocks. |
| 20:17 | - | Reserved |
| 21 | PDN | 1: The A/D converter is operational.  0: The A/D converter is in power-down mode. |
| 23:22 | - | Reserved |
| 26:24 | START | When the BURST bit is 0, these bits control whether and when an A/D conversion is started:  000: No start (this value should be used when clearing PDN to 0).  001: Start conversion now.  010: Start conversion when the edge selected by bit 27 occurs on the P2.10 / EINT0 / NMI pin. Set the pin function to EINT0 in PINSEL4 register.  011: Start conversion when the edge selected by bit 27 occurs on the P1.27 / CLKOUT / USB\_OVRCRn / CAP0.1 pin. Set the pin function to CAP0.1 in PINSEL3 register.  100: Start conversion when the edge selected by bit 27 occurs on MAT0.1. Note that this does not require that the MAT0.1 function appear on a device pin.  101: Start conversion when the edge selected by bit 27 occurs on MAT0.3. Note that it is not possible to cause the MAT0.3 function to appear on a device pin.  110: Start conversion when the edge selected by bit 27 occurs on MAT1.0. Note that this does not require that the MAT1.0 function appear on a device pin.  111: Start conversion when the edge selected by bit 27 occurs on MAT1.1. Note that this does not require that the MAT1.1 function appear on a device pin. |
| 27 | EDGE | This bit is significant only when the START field contains 010-111. In these cases:  1: Start conversion on a falling edge on the selected CAP/MAT signal.  0: Start conversion on a rising edge on the selected CAP/MAT signal. |
| 31:28 | - | Reserved |

* + 1. ADINTEN register:

ADINTEN (A/D Interrupt Enable) register bit description (From Table 534, page 589 in “User Manual”).

|  |  |  |
| --- | --- | --- |
| **Bit** | **Symbol** | **Description** |
| 0 | ADINTEN0 | 1: Completion of conversions on ADC channel 0 will generate an interrupt  0: Completion of conversions on ADC channel 0 won’t generate an interrupt |
| 1 | ADINTEN1 | 1: Completion of conversions on ADC channel 1 will generate an interrupt  0: Completion of conversions on ADC channel 1 won’t generate an interrupt |
| 2 | ADINTEN2 | 1: Completion of conversions on ADC channel 2 will generate an interrupt  0: Completion of conversions on ADC channel 2 won’t generate an interrupt |
| 3 | ADINTEN3 | 1: Completion of conversions on ADC channel 3 will generate an interrupt  0: Completion of conversions on ADC channel 3 won’t generate an interrupt |
| 4 | ADINTEN4 | 1: Completion of conversions on ADC channel 4 will generate an interrupt  0: Completion of conversions on ADC channel 4 won’t generate an interrupt |
| 5 | ADINTEN5 | 1: Completion of conversions on ADC channel 5 will generate an interrupt  0: Completion of conversions on ADC channel 5 won’t generate an interrupt |
| 6 | ADINTEN6 | 1: Completion of conversions on ADC channel 6 will generate an interrupt  0: Completion of conversions on ADC channel 6 won’t generate an interrupt |
| 7 | ADINTEN7 | 1: Completion of conversions on ADC channel 7 will generate an interrupt  0: Completion of conversions on ADC channel 7 won’t generate an interrupt |
| 8 | ADGINTEN | 0: Only the individual ADC channels enabled by ADINTEN7:0 will generate interrupts.  Remark: This bit must be set to 0 in burst mode (BURST = 1 in the AD0CR register)  1: Only the global DONE flag in ADDR is enabled to generate an interrupt. |
| 31:9 | - | Reserved |

* + 1. ADSTAT register:

ADSTAT (A/D Status) register bit description (From Table 536, page 591 in “User Manual”).

|  |  |  |
| --- | --- | --- |
| **Bit** | **Symbol** | **Description** |
| 0 | DONE0 | This bit mirrors the DONE status flag from the result register for A/D channel 0. |
| 1 | DONE1 | This bit mirrors the DONE status flag from the result register for A/D channel 1. |
| 2 | DONE2 | This bit mirrors the DONE status flag from the result register for A/D channel 2. |
| 3 | DONE3 | This bit mirrors the DONE status flag from the result register for A/D channel 3. |
| 4 | DONE4 | This bit mirrors the DONE status flag from the result register for A/D channel 4. |
| 5 | DONE5 | This bit mirrors the DONE status flag from the result register for A/D channel 5. |
| 6 | DONE6 | This bit mirrors the DONE status flag from the result register for A/D channel 6. |
| 7 | DONE7 | This bit mirrors the DONE status flag from the result register for A/D channel 7. |
| 8 | OVERRUN0 | This bit mirrors the OVERRRUN status flag from the result register for A/D channel 0. |
| 9 | OVERRUN1 | This bit mirrors the OVERRRUN status flag from the result register for A/D channel 1. |
| 10 | OVERRUN2 | This bit mirrors the OVERRRUN status flag from the result register for A/D channel 2. |
| 11 | OVERRUN3 | This bit mirrors the OVERRRUN status flag from the result register for A/D channel 3. |
| 12 | OVERRUN4 | This bit mirrors the OVERRRUN status flag from the result register for A/D channel 4. |
| 13 | OVERRUN5 | This bit mirrors the OVERRRUN status flag from the result register for A/D channel 5. |
| 14 | OVERRUN6 | This bit mirrors the OVERRRUN status flag from the result register for A/D channel 6. |
| 15 | OVERRUN7 | This bit mirrors the OVERRRUN status flag from the result register for A/D channel 7. |
| 16 | ADINT | This bit is the A/D interrupt flag. It is one when any of the individual A/D channel Done flags is asserted and enabled to contribute to the A/D interrupt via the ADINTEN register. |
| 31:17 | - | Reserved |

* + 1. ADGDR register:

ADGDR (A/D Global Data) register bit description (From Table 533, page 589 in “User Manual”).

|  |  |  |
| --- | --- | --- |
| **Bit** | **Symbol** | **Description** |
| 3:0 | - | Reserved |
| 15:4 | RESULT | When DONE is 1, this field contains a binary fraction representing the voltage on the AD0[n] pin selected by the SEL field, as it falls within the range of VREFP to VREFN. Zero in the field indicates that the voltage on the input pin was less than, equal to, or close to that on VREFN, while 0xFFF indicates that the voltage on the input was close to, equal to, or greater than that on VREFP. |
| 23:16 | - | Reserved |
| 26:24 | CHN | These bits contain the channel from which the RESULT bits were converted (e.g. 000 identifies channel 0, 001 channel 1...). |
| 29:27 | - | Reserved |
| 30 | OVERRUN | This bit is 1 in burst mode if the results of one or more conversions was (were) lost and overwritten before the conversion that produced the result in the RESULT bits. This bit is cleared by reading this register. |
| 31 | DONE | This bit is set to 1 when an A/D conversion completes. It is cleared when this register is read and when the ADCR is written. If the ADCR is written while a conversion is still in progress, this bit is set and a new conversion is started. |

* + 1. ADDR0 to ADDR7 registers:

ADDR0 to ADDR7 (A/D Data) register bit description (From Table 535, page 590 in “User Manual”).

|  |  |  |
| --- | --- | --- |
| **Bit** | **Symbol** | **Description** |
| 3:0 | - | Reserved |
| 15:4 | RESULT | When DONE is 1, this field contains a binary fraction representing the voltage on the AD0[n] pin, as it falls within the range of VREFP to VREFN. Zero in the field indicates that the voltage on the input pin was less than, equal to, or close to that on VREFN, while 0xFFF indicates that the voltage on the input was close to, equal to, or greater than that on VREFP. |
| 29:16 | - | Reserved |
| 30 | OVERRUN | This bit is 1 in burst mode if the results of one or more conversions was (were) lost and overwritten before the conversion that produced the result in the RESULT bits. This bit is cleared by reading this register. |
| 31 | DONE | This bit is set to 1 when an A/D conversion completes. It is cleared when this register is read. |

### **III. LAB PROCERUCE**

III.1 Task 1

* Requirement:

Create “ADC.c” and “ADC.h” files that configure ADC module. *(On the evaluation board, the ADC function connecting to the Rotary Knob is AD0.2 function).*

* Have an initialization function: “ADC\_Initialize()”
* A function to start A/D conversion: “ADC\_StartConversion()”
* A function to get whether ADC process is done: “ADC\_ConversionDone()”
* A function to get the last ADC value: “ADC\_GetValue()”

Write the main code that:

* Process ADC, then display the value to 8-leds, ( the binary value of 8 high order bit will be display in 8 LED)

III.2 Task 2

* Requirement:

Create “Uart.c” and “Uart.h” files that configure UART Prototype that have:

* An initialization function: “void UART\_Initialize(unsigned int baudRate)” that can set UART to run at an inputted baud rate.
* A function to send data: “UART\_Send(char \*bufferPtr, unsigned int length)”
* Have a buffer variable “unsigned char UART\_Buffer[BUFFER\_SIZE]” to stored received data. When you want to read received data, read from this variable.
* Have a variable “unsigned int UART\_Buffer\_Count” to determine how many data is stored in the buffer.

Write the main code that:

* When received “0xFF”: turn all LEDs on and send “LEDs are turned on” back to PC.
* When received “0xF0”: turn all LEDs off and send “LEDs are turned off” back to PC.

Follows the guide in Lab 2 – Tutorial.

III.3 Task 3

* Requirement:

Use “Uart.c” and “Uart.h” from Task 1.

Write the main code that:

* Sending data to PC bases on joystick status:
* Send “International University” to PC through UART when Joystick is pulled up.
* Send “Electrical Engineering” to PC through UART when Joystick is pulled down.
* Send “Embedded System” to PC through UART when Joystick is pulled to the left.
* Send your name to PC through UART when Joystick is pulled to the right.
* Turning LEDs on/off bases on data received from PC:
* Each byte received from PC represent a request that has 2 parts:

4 most significant bits represent the intention (1 to turn on, 0 to turn off).

4 lest significant bits represent the target (the index of the led).

* For example:

*When PC send 0x17, the 7th led is on. When PC send 0x27, the 7th led is off.*

*When PC send 0x1A or 0x27, nothing happens.*

III.4 Task 4

* Requirement:

Copy Task 2 solution and improve it.

Create “ADC.c” and “ADC.h” files that configure ADC module. *(On the evaluation board, the ADC function connecting to the Rotary Knob is AD0.2 function).*

* Have an initialization function: “ADC\_Initialize()”
* A function to start A/D conversion: “ADC\_StartConversion()”
* A function to get whether ADC process is done: “ADC\_ConversionDone()”
* A function to get the last ADC value: “ADC\_GetValue()”

Write a function called UintToChar that converts int to char\*.

Write the main code that:

* Process ADC, send that value as a string to PC.
* Sending data to PC bases on joystick status (Same as Task 2):
* Send “International University” to PC through UART when Joystick is pulled up.
* Send “Electrical Engineering” to PC through UART when Joystick is pulled down.
* Send “Embedded System” to PC through UART when Joystick is pulled to the left.
* Send your name to PC through UART when Joystick is pulled to the right.
* Turning LEDs on/off bases on data received from PC (Same as Task 2):
* Each byte received from PC represent a request that has 2 parts:

4 most significant bits represent the intention (1 to turn on, 0 to turn off).

4 lest significant bits represent the target (the index of the led).

* For example:

*When PC send 0x17, the 7th led is on. When PC send 0x27, the 7th led is off.*

*When PC send 0x1A or 0x27, nothing happens.*

**IV . LAB TUTORIAL**

IV.1 Task 1:

Use “Led.c”, “Led.h”, “Joystick.c” and “Joystick.h” from Lab 2 are used for tasks in Lab 4

“adc.h” code:

extern void ADC\_Initialize (void);

extern void ADC\_StartConversion (void);

extern int ADC\_ConversionDone (void);

extern int ADC\_GetValue (void);

The code bellow is incomplete, from the hints in comments and refer to corresponding register bit description, complete the blank to make it work.

ADC Initialize code in “adc.c” file:

#include "LPC17xx.h"

#include "ADC.h"

// Last converted value

static unsigned int AD\_lastValue;

// AD conversion done flag

static unsigned int AD\_done;

void ADC\_Initialize (void)

{

// Enable power to ADC & IOCON

LPC\_SC->PCONP \_\_\_\_\_\_\_\_\_\_\_\_\_\_;

// Set PINSEL1[19:18] = 01 to select Function AD0.2 for pin P0.25

LPC\_PINCON->PINSEL1 \_\_\_\_\_\_\_\_\_\_\_\_\_\_;

LPC\_PINCON->PINSEL1 \_\_\_\_\_\_\_\_\_\_\_\_\_\_;

// Select AD0.2 pin, ADC clock is 25MHz/5, Enable ADC

LPC\_ADC->ADCR \_\_\_\_\_\_\_\_\_\_\_\_\_\_;

// Enable Global ADC interrupt

LPC\_ADC->ADINTEN \_\_\_\_\_\_\_\_\_\_\_\_\_\_;

// Enable ADC Interrupt in the NVIC interrupt controller

NVIC\_EnableIRQ(ADC\_IRQn);

}

void ADC\_IRQHandler(void)

{

// Read ADC clears interrupt

int adstat = LPC\_ADC->ADSTAT;

// Store converted value

AD\_lastValue \_\_\_\_\_\_\_\_\_\_\_\_\_\_;

AD\_done = 1;

}

void ADC\_StartConversion (void)

{

// Stop conversion

LPC\_ADC->ADCR \_\_\_\_\_\_\_\_\_\_\_\_\_\_;

// Start conversion

LPC\_ADC->ADCR \_\_\_\_\_\_\_\_\_\_\_\_\_\_;

}

int ADC\_ConversionDone (void)

{

return AD\_done;

}

int ADC\_GetValue (void)

{

return AD\_lastValue;

}

\\ main program example

#include "LPC17xx.h"

#include "leds.h"

#include "adc.h"

int main()

{

leds\_init();

ADC\_Initialize();

int ADC\_Value;

char led\_value;

while(1)

{

ADC\_StartConversion();

while(ADC\_ConversionDone()==0);

ADC\_Value = ADC\_GetValue();

led\_value = (char) ADC\_Value;

leds\_set\_value(led\_value);

}

}

IV.2 Task 2

* Requirement:

Create “Uart.c” and “Uart.h” files that configure UART Prototype that have:

* An initialization function: “void UART\_Initialize(unsigned int baudRate)” that can set UART to run at an inputted baud rate.
* A function to send data: “UART\_Send(char \*bufferPtr, unsigned int length)”
* Have a buffer variable “unsigned char UART\_Buffer[BUFFER\_SIZE]” to stored received data. When you want to read received data, read from this variable.
* Have a variable “unsigned int UART\_Buffer\_Count” to determine how many data is stored in the buffer.

Write the main code that:

* When received “0xFF”: turn all LEDs on and send “LEDs are turned on” back to PC.
* When received “0xF0”: turn all LEDs off and send “LEDs are turned off” back to PC.
* Guide:
* “Uart.h” Code:

Define the bit position in require registers.

#define IER\_EnableRBR 0x01

#define IER\_EnableTHRE 0x02

#define IER\_EnableRLS 0x04

#define IIR\_ReceiveLineStatus 0x03

#define IIR\_ReceiveDataAvailable 0x02

#define IIR\_CharacterTimeoutIndicator 0x06

#define IIR\_THREInterrupt 0x01

#define LSR\_RecieverDataReady 0x01

#define LSR\_OverrunError 0x02

#define LSR\_ParityError 0x04

#define LSR\_FramingError 0x08

#define LSR\_BreakInterrupt 0x10

#define LSR\_TransmitterHoldingRegisterEmpty 0x20

#define LSR\_TransmitterEmpty 0x40

#define LSR\_ErrorInRxFifo 0x80

#define BUFFER\_SIZE 0x40

extern unsigned int UART\_Buffer\_Count;

extern unsigned char UART\_Buffer[BUFFER\_SIZE];

extern void UART\_Initialize(unsigned int baudrate);

extern void UART\_Send(char \*bufferPtr, unsigned int length);

The code bellow is incomplete, from the hints in comments and refer to corresponding register bit description, complete the blank to make it work.

* UART Initialize code in “Uart.c” file:

#include "LPC17xx.h"

#include "uart.h"

unsigned int UART\_Buffer\_Count;

unsigned char UART\_Buffer[BUFFER\_SIZE];

char UART\_TX\_Is\_Empty = 1;

void UART\_Initialize(unsigned int baudRate)

{

unsigned int divisorLatchRate;

unsigned int peripheralClock;

// Enable UART0 Power/Clock

LPC\_SC->PCONP \_\_\_\_\_\_\_\_\_\_\_\_\_\_;

// Setting PINSEL0[7:4] to [0101] to use P0.2 for TXD0, P0.3 for RXD0

LPC\_PINCON->PINSEL0 \_\_\_\_\_\_\_\_\_\_\_\_\_\_;

LPC\_PINCON->PINSEL0 \_\_\_\_\_\_\_\_\_\_\_\_\_\_;

// Set UART0 Peripheral Clock = CCLK/4

LPC\_SC->PCLKSEL0 \_\_\_\_\_\_\_\_\_\_\_\_\_\_;

// 8-bit length, 1 stop bit, No Parity

// Enable access to Divisor Latches

LPC\_UART0->LCR \_\_\_\_\_\_\_\_\_\_\_\_\_\_;

// Let DivAddVal be 0 & MulVal be 1 as default -> not set FDR register

// Calculate Peripheral Clock

peripheralClock = SystemCoreClock/4;

// Calculate Divisor Latch Rate, which is 16 \* Baud rate

divisorLatchRate = (peripheralClock/16)/baudRate;

// Set value for Divisor Latch MSB register from divisorLatchRate

LPC\_UART0->DLM = \_\_\_\_\_\_\_\_\_\_\_\_\_\_;

// Set value for Divisor Latch LSB register from divisorLatchRate

LPC\_UART0->DLL = \_\_\_\_\_\_\_\_\_\_\_\_\_\_;

// Disable access to Divisor Latches

LPC\_UART0->LCR \_\_\_\_\_\_\_\_\_\_\_\_\_\_;

// Enable FIFO, and reset TX and RX FIFO

// Set RX Trigger on 1 character recieved

LPC\_UART0->FCR \_\_\_\_\_\_\_\_\_\_\_\_\_\_;

// Enable UART0 interrupts

LPC\_UART0->IER = IER\_EnableRBR | IER\_EnableTHRE | IER\_EnableRLS;

// Enable UART0 Interrupt in the NVIC interrupt controller

NVIC\_EnableIRQ(UART0\_IRQn);

}

* Handle UART Interrupts code in “Uart.c” file:

Get “Interrupt Identification” than handle that interrupts.

void UART0\_IRQHandler (void)

{

int InterruptIdRegister, LineStatusRegister;

int interruptId;

// Read Interrupt Identification Register

InterruptIdRegister = LPC\_UART0->IIR;

// Get [3:1] bits indicating the Interrupt Identification

interruptId \_\_\_\_\_\_\_\_\_\_\_\_\_\_;

// RLS (Receive Line Status) Interrupt

if (interruptId == IIR\_ReceiveLineStatus)

{

// Not handle these errors in this lab

// Read Line Status Register will clear the interrupt

LineStatusRegister \_\_\_\_\_\_\_\_\_\_\_\_\_\_;

}

// RDA (Receive Data Available) Interrupt

else if (interruptId == IIR\_ReceiveDataAvailable)

{

// Get data from UART Reciever Buffer Register

UART\_Buffer[UART\_Buffer\_Count] \_\_\_\_\_\_\_\_\_\_\_\_\_\_;

UART\_Buffer\_Count++;

if (UART\_Buffer\_Count == BUFFER\_SIZE)

{

// Buffer is overflow now

UART\_Buffer\_Count = 0;

}

}

// THRE (Transmit Holding Register Empty) Interrupt

else if (interruptId == IIR\_THREInterrupt)

{

// Transmitter Holding Register is empty now

UART\_TX\_Is\_Empty = 1;

}

// CTI (Character Timeout Indicator) Interrupt will never occur since

// we rise "RDA (Receive Data Available) Interrupt"

// every byte we received.

}

* Send Function code in “Uart.c” file:

When this function is called, send ‘length’ char from the pointer ‘bufferPtr’.

void UART\_Send(char \*bufferPtr, unsigned int length)

{

while ( length != 0 )

{

// Wait until Transmission is done

while (UART\_TX\_Is\_Empty \_\_\_\_\_\_\_\_\_\_\_\_\_\_);

LPC\_UART0->THR = \*bufferPtr;

// Transmitter Holding Register is not empty now

UART\_TX\_Is\_Empty \_\_\_\_\_\_\_\_\_\_\_\_\_\_;

bufferPtr++;

length--;

}

}

* Main code:

If UART buffer has data, read the last byte and turn LEDs on/off base on it. Then send back a string to PC, and set buffer byte-counting to 0.

#include "cmsis\_os.h" // ARM::CMSIS:RTOS:Keil RTX

#include "LPC17xx.h" // Device header

#include "Uart.h"

#include "Led.h"

int main(void)

{

UART\_Initialize(9600);

Led\_Initialize();

while(1)

{

if(UART\_Buffer\_Count>0)

{

if(UART\_Buffer[UART\_Buffer\_Count-1] == 0xFF)

{

Leds\_Set\_Value(0xFF);

UART\_Send("\nLEDs are turned on",19);

UART\_Buffer\_Count = 0;

}

else if(UART\_Buffer[UART\_Buffer\_Count-1] == 0xF0)

{

Leds\_Set\_Value(0x00);

UART\_Send("\nLEDs are turned off",20);

UART\_Buffer\_Count = 0;

}

}

}

}

IV.3 Task 3

* Requirement:

Use “Uart.c” and “Uart.h” from Task 1.

Write the main code that:

* Sending data to PC bases on joystick status:
* Send “International University” to PC through UART when Joystick is pulled up.
* Send “Electrical Engineering” to PC through UART when Joystick is pulled down.
* Send “Embedded System” to PC through UART when Joystick is pulled to the left.
* Send your name to PC through UART when Joystick is pulled to the right.
* Turning LEDs on/off bases on data received from PC:
* Each byte received from PC represent a request that has 2 parts:

4 most significant bits represent the intention (1 to turn on, 0 to turn off).

4 lest significant bits represent the target (the index of the led).

* For example:

*When PC send 0x17, the 7th led is on. When PC send 0x27, the 7th led is off.*

*When PC send 0x1A or 0x27, nothing happens.*

IV.4 Task 4

* Requirement:

Copy Task 2 solution and improve it.

Create “ADC.c” and “ADC.h” files that configure ADC module. *(On the evaluation board, the ADC function connecting to the Rotary Knob is AD0.2 function).*

* Have an initialization function: “ADC\_Initialize()”
* A function to start A/D conversion: “ADC\_StartConversion()”
* A function to get whether ADC process is done: “ADC\_ConversionDone()”
* A function to get the last ADC value: “ADC\_GetValue()”

Write a function called UintToChar that converts int to char\*.

Write the main code that:

* Process ADC, send that value as a string to PC.
* Sending data to PC bases on joystick status (Same as Task 2):
* Send “International University” to PC through UART when Joystick is pulled up.
* Send “Electrical Engineering” to PC through UART when Joystick is pulled down.
* Send “Embedded System” to PC through UART when Joystick is pulled to the left.
* Send your name to PC through UART when Joystick is pulled to the right.
* Turning LEDs on/off bases on data received from PC (Same as Task 2):
* Each byte received from PC represent a request that has 2 parts:

4 most significant bits represent the intention (1 to turn on, 0 to turn off).

4 lest significant bits represent the target (the index of the led).

* For example:

*When PC send 0x17, the 7th led is on. When PC send 0x27, the 7th led is off.*

*When PC send 0x1A or 0x27, nothing happens.*

**IV. LAB PERFORMANCE GRADING AND LAB REPORT GUIDELINES**

For each Lab experiment Students show the successful running results to Lab Instructor for Lab Performance grading.

Students write a report which includes : Algorithm flowchart and C++ Code for each experiment. In each block of the code or line of code, give the comments for the meaning of this block of code.