CpE5160 Experiment #2
UART Interface

Introduction

The purpose of this experiment is to initialize the UART, which will be used for debugging in later experiments. Some functions are given to the programmer to display the contents of memory as hexadecimal values and as ASCII characters. This will help the programmer locate items stored on the SD card. The file organization tips given in the book “Embedded C” should be used to create reusable code for later experiments.

Serial Communication

The UART will be used as an RS232 port in a later part of the project to view blocks of information from the SD card. This will allow the programmer to see the properties of the SD card and to study the FAT file system used to store information on the SD card.

The following paragraphs describe the software routines that will be needed to initialize the UART and print information from the SD card in a useful format.

UART Initialization

The first function needed is an initialization routine for the UART. This initialization function should be written so that it can be used to initialize each of the UARTs in the microcontroller. This is a device driver function and should be in a file with the other device driver functions for the UART. The baud rate should be set first. The equation for setting the baud rate can be found in the datasheet and in the lecture notes. This calculation should use the CPU oscillator frequency (F_CPU) and oscillator divider (OSC_DIV) to calculate the reload value to generate the baud rate. The UART mode, number of data bits, parity, and number of stop bits are selected next. Once these are values are set, then the transmitter and/or receiver can be enabled. See the datasheet and lecture notes for setting all these values.

Transmit and Receive Functions

The transmit function has two input parameters, the base address of the USART to use and the 8-bit value to send. The function should first check to see if the USART Data Register Empty (UDRE) bit is set. If it is not set, this means a value is in USART Data Register (UDR) and a new value cannot be written, or it will overwrite the previous value (a transmit overrun). UDRE will be set to 1 by hardware when the UDR is empty. The UDRE bit is a read-only bit in the UCSRA register. It should be read and the UDRE bit isolated using a bitwise AND. This should be repeated until UDRE is ‘1’ (Note that UCSRA must read each time through the loop, or you could end up in an endless loop). Since the UDRE bit is 5 of the register, when it is set, the bit-wise AND will not result in 0x01, but 0x20 (1<<5). After UDRE is set to ‘1,’ then the input value can be written to UDR. Some systems use error checking by returning the value that was sent or some error value. The function described does not perform any error checking. The programmer may wish to add a simple timeout to prevent an endless loop if UDRE is never set and then an error could be returned for this condition.

The receive function only has one input parameter, the base address of the USART, and it returns the received value. This function waits until the RXC bit is set. The RXC bit is a read-only bit in the UCSRA register and is set by hardware when a value is received. The received value should then be read from UDR before the next value is received (a receive overrun error). The RXC bit is cleared by hardware when the value from the UDR is read. The value read from the UDR should be returned by the function. This type of function is a blocking function because it halts code execution until a value is received. This will work for most of the project since nothing else is happening when waiting for the user to input information. The solution code will mostly implement this function.

An alternate version of the receive function can implement some error checking and possibly be written in a non-blocking manner which can allow other tasks to be executed while waiting for user input. The return value could be a flag that indicates if a value is received or not and if an error has occurred (possible values: no_value, good_value, parity_error, rcv_overrun). If the RXC bit was not set, then the return value would be no_value. If it was set and no errors have occurred, then good_value is returned along with the received value. A pointer to a variable from the calling function can be used to return the received value if one has been received. The parity error (UPE) and data overrun (DOR) bits would also be checked along with the RXC bit to determine the other possible return flag values. The calling function must check the return value and only process the received value if the flag is good_value.

UART_Print Functions
The functions described in the following sections are provided with the experiment #2 project files. They should not be changed. If your UART_Transmit and UART_Receive functions do not match with the function call statements in these files, then you will need to change your source code or use a wrapper function to accommodate the differences.

A static global array called print_buffer that is 80 characters long is used to hold a string of characters to be printed. The base address of this array can be imported into a function by calling the Export_print_buffer() function and storing the return value into a char pointer. The function Copy_String_to_Buffer(), which copies a string of bytes from Flash memory into SRAM is provided. The following parameters are needed for this function: the number of bytes (0 will copy characters until a NULL character is encountered i.e., a complete NULL terminated string), the starting Flash memory address and the starting SRAM address.

The UART_Transmit_String() function will send a string of characters using the specified UART. The function needs three parameters: The base address of the UART, the number of bytes to send (0 the send characters until a NULL character is encountered) and the starting SRAM address of the string to send.

Printing Byte Values from Memory

The following functions are given to the student in the print_memory.c and print_memory.h source code files. These functions can serve as an example of how to declare strings in Flash memory and print them using the UART_Print functions. The print memory function has two input parameters, the starting memory address of the block of memory to print and the number of bytes of memory to print. The function will print enough lines of sixteen bytes to print all the bytes requested plus the extra to make a full sixteen-byte line. The destination UART is defined in the print_memory.h file as the standard output (std_out).

The sprintf() function in the stdio.h library is used to print values, however some adjustments are sometimes needed for accurate printing. The method of printing a byte in hexadecimal described here can be used for later experiments. The *printf* formatter %X can format the value in hexadecimal. If the value is a single digit, this can sometimes throw off the column alignment, so a width and a precision specification can also be added. A width specification forces the output to be at least this minimum width. The precision specification will force zeros to be added to the left of an integer to fill the minimum precision specification. For example, the width and precision specification is written as width.precision. A byte will always be printed as two hexadecimal digits so a width specification of 2 is used. So a formatter of "%2X" will print a 0 value as " 0" and a formatter of "%2.2X" will print a 0 value as "00." A space should be placed after the formatter to print a space between each byte that is printed.

A private function is used that prints 16 bytes on one line. The function is passed a generic pointer to point to the first byte of the line to be printed. This pointer can be printed using the formatter %p which will print the memory type as a letter and colon followed by a 16-bit address value. As an alternative the pointer can be converted to an integer (var16=(uint16) address_p;) and printed as an integer. The function can use the printf formatter described in the previous paragraph to print the 16 bytes in hexadecimal. On the same line, those same 16 bytes are also printed as ASCII characters. If they are not printable characters, a period is printed instead. Printable ASCII characters have values between 0x20 (32) and 0x7E (126). Printing bytes in this manner will allow the user to see specific values or ASCII strings located in memory.

A memory display function, print_memory(), is needed for the next part of the project. It is passed the starting address as a pointer and the number of bytes to print as an unsigned int. The previous private function is used to print each line. This function can be tested by placing a string of text in memory and printing the memory values. For example, the statement: char str_name[] = "Hello World!"; creates a string in memory. The address of that string in memory is the pointer "*str_name*." Using this as the starting address, the programmer should be able to view the contents in memory containing the string. In a later part of the project, this routine will be used to view the contents of the SD card.

Procedure

The following equipment will be required for this part of the experiment:

- ATMEGA324PB Xplained development board
- Micro-B USB Cable
- Files from Canvas:
  - print_memory.c
  - print_memory.h
  - UART_print.c
  - UART_print.h
- PC with
  - Atmel Studio 7
  - Optional Terminal Program (For Example: Putty)

1) (4pts) Create a new executable C project for this experiment. This will automatically create a main.c source code file. Any project should have a board.h file to describe your hardware. Make sure the defined values F_CPU and OSC_DIV match with your circuit.

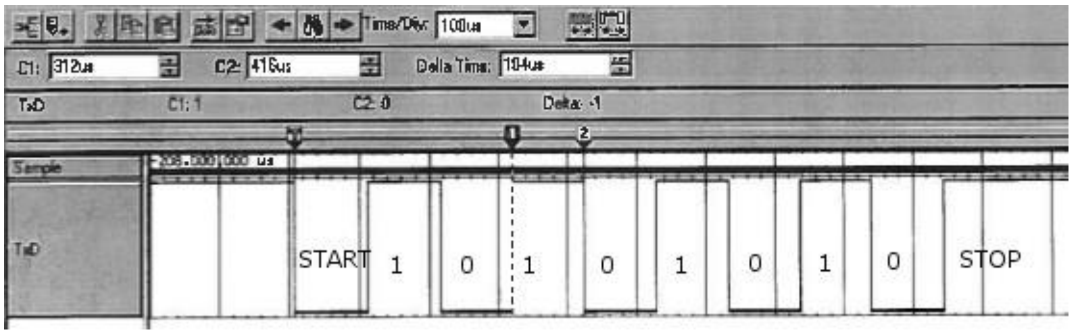
2) (10pts) Create a new header file and C source code file to contain the device drivers for the UART (UART_init(), UART_transmit() and UART_receive() functions).

   Write an initialization routine for the UART: *uint8_t UART_init(volatile uint8_t *UART_addr, uint32_t baud_rate);*. This routine should use a calculation to determine the baud rate generator reload values from a baud rate value passed as a parameter to the function. The equation for the baud rate generator calculation can be found in the ATMEGA324PB datasheet and in the lecture notes. The initialization function also needs to place the correct values in UCSRA, UCSRC and UCSRB. Based on the restrictions of the simple embedded operating system (sEOS) used in the book "Embedded C", the UART interrupts should be disabled. Make sure the settings can be clearly read by another programmer (well documented).

3) (6pts) Create the *uint8_t UART_transmit(volatile uint8_t *UART_addr, uint8_t send_value);* and the *uint8_t UART_receive(volatile uint8_t *UART_addr);* functions described in the background material. The optional function for a non-blocking receive function can also be created for a few bonus points: *uint8_t UART_receive_nb(volatile uint8_t *UART_addr, uint8_t *rcvd_value);*

4) Debugging: UART1 is connected to the embedded debugger and will send and receive characters through a virtual COM port on the PC using the data visualizer in Microchip Studio, Putty or another terminal program. UART0 is connected to the TX and RX pins on the development board and can be used to measure the baud rate with the oscilloscope. Call the UART_init() function in the

initialization area of the main function to initialize UART1. I recommend the UART settings of 9600 8-N-1 because they are the default settings for Putty. Place the UART_transmit() function in the endless while(1) loop with some LEDs that flash at some rate between 0.5 to 1 seconds. This will create a delay that will control how fast the characters are transmitted. The delay can be removed to test if the UART_transmit function is checking the UDRE bit correctly. Send a character to verify that the UART is functioning properly. I recommend the ASCII character 'U' (0x55) because of its alternating '1' and '0' pattern. For debugging, use UART0 and an oscilloscope or a logic analyzer to measure your baud rate at the TX pin on the development board. The pulse width of one bit should be the inverse of the baud rate. For a baud rate of 9600, this would be about 104µs. The figure below shows an ASCII 'U' as measured with a logic analyzer. Text was added to show each part of the expected waveform. The measured time is hard to read, but it shows a pulse width of 104µs.



ASCII 'U' measured on TxD pin (PD3)

5) (2pts) Place the UART_receive function in the while(1) loop where it will wait until a character is received. Use the UART_transmit function to echo this character back to the Putty terminal program. Verify that these two functions are working correctly. The LED functions and the _delay_ms() functions can be commented out for a faster response to characters being entered (for grading, please do not delete them). For a few bonus points, you can implement the non-blocking receive function that allows the LEDs to continue to blink and calls UART_transmit only if a character is received.

6) (2pts) Add the UART_print.c source code to the project. In the initialization area of the project, import the print buffer pointer from the UART_print source code using the export_print_buffer() function. Add the stdio.h library to the main.c source code. Add an sprintf() statement that places a string of characters into the print buffer. The sprintf() function requires a pointer to a char array and the print_buffer is declared as a char array. However, some of the other functions use a uint8_t pointer. You can use typecasting, such as (uint8_t *), to change a char pointer to a uint8_t pointer. This will not matter when using ASCII characters since they are just values for characters and have no sign. Send these using the UART_transmit_string() function. If you examine the SRAM using the debugger's memory window, you should be able to see the sprintf() string stored in SRAM before it is placed in the print buffer.

7) (3pts) Create an array of characters (a string) that are stored in Flash memory. This will require the library file <avr/pgmspace.h>. Declare a char array as const with the PROGMEM macro:
    const char string_name[15] PROGMEM = {"Hello World!\n\r\0"};
   Refer to the print memory source code for an example of declaring a string in Flash memory. In the initialization area, use the copy_string_to_buffer() function to copy this string to the print buffer and then transmit it using the UART_transmit_string() function.

8) (3pts) Add the print_memory.c source code file to the project and use #include statements to include the header file print_memory.h in your C source code. Declare a string in memory that is at least 30 characters long and includes the names of the members of your group. Call print memory function to print the memory area that contains the string you have declared (use the name of the string as a pointer to this memory area).

9) Include a screen capture of the Putty output showing the printed strings and the memory printout with your submitted source code. Place your project in a compressed folder (zip file) and post it to canvas or e-mail it to me at youngerr@mst.edu. Make sure that if you are working with a group, place your names in the comment headers. Only one person in the group needs to post the source code files.

Grading:

Functionality:	Code works as described in the assignment: Registers set as needed and actions occur as specified; Questions answered and calculations shown.	Points awarded as specified in the steps marked graded. (30 points)
Compile Errors:	If submitted code does not compile, then a minimum of 25% of the functionality points will be deducted. More may be deducted for multiple errors.	Minimum of 25% of functionality point deducted.
Bonus	UART_receive_nb function and implementation	5 points
Organization:	Seperating device drivers (UART) and application code.	10 points
Readability:	Source code is well commented. Descriptive names are used for functions, constants, and variables.	10 points
Correctness	No patches or workarounds are used in the source code to get the correct functionality. Basic code standard such as mentioned in the background documents is followed.	10 points