**Reading and Displaying UART Data Using Raspberry Pi and EMF8BB3**

Camille Chow and Ostap Voynarovski

***Abstract – Using the EFM8 Busy Bee microcontroller and a Raspberry Pi, a system was designed to take the keystrokes from a USB keyboard and display the characters, either in normal or hex mode, onto an LCD display. This project required the use of the EFM8’s UART and the Raspberry Pi’s serial data. Software was coded in SimplicityStudio in order to write to the display. Although the final implementation lacked certain requirements, it was successful in interfacing with the UART and displaying keystrokes to the LCD.***

### I. Introduction

One of the main aspects of this project involved receiving and transmitting data from the EMF8’s universal asynchronous receiver/transmitter (UART). The UART is used to communicate with serial data, and the speed at which the data is transferred is controlled by the baud rate, the number of bits per second. For the purposes of this project, the baud rates were set to 115200, allowing for very fast data transmission.

This transmission of serial data requires a shift register clocked by a frequency related to the baud rate. When receiving data, the UART receiver tests the state of the incoming signal on each clock pulse, looking for a start bit for a character or other data type. Once the data bit is detected, the contents of the shift register are made available to the receiving system and an interrupt is thrown by the UART, indicating that the data is available. This process repeats until a stop bit is encountered.

When transmitting, as soon as the transmitting system places a value in the shift register, the UART generates a start bit, shifts the required number of data bits to the output, and sends the stop bit.

For the EFM8 UART, which can receive and transmit at the same time, two shift registers are required. Because shift registers are used, the data is transmitted and read in a first-in, first-out fashion.

The use of registers in the EFM8 allows fast memory access, as they are located closer to the processor in memory hierarchy. However, this fast access is achieved through the sacrifice of size; registers typically do not store a lot of data compared to other types of memory such as RAM.

Other EFM8 registers were utilized in this project as well, including those used to store the data from the screen buffer, which is described in the design section. An additional interrupt control register is used to generate interrupt flags.

When such interrupts are thrown, this affects the pipelining in the processor. While executing instructions, the microcontroller uses pipelining to process multiple instructions at the same time, in a staggered timeline. The interrupt disrupts this process, clearing the pipeline in order to run the interrupt service routine.

The final part of this project was the implementation of a “bare metal” code in the ARM assembly language to transmit keystrokes received from the USB port of the Raspberry Pi to the serial port. This low level bare metal code interacts with the Pi at a hardware level, without the use of the operating system.

This assignment required the use of many of the EFM8 and Raspberry Pi peripherals, as well as a few I/O devices. The peripherals required on the EFM8 included the built-in timers, push buttons, LCD screen, joystick, and UART. The Raspberry Pi serial port was required as well. Interfacing between the UART and the USB on the personal computer we used was done using an FTDI cable with transmit/receive pins corresponding to those on the EFM8. Interfacing between the Pi and the EFM8 required jumper wires that connected the receive on the serial port to the transmit on the UART and vice versa. The USB keyboard was meant to connect to the Raspberry Pi’s USB port, and its data was meant to be sent to the serial using the bare metal code.

Through the integration of this hardware with SimplicityStudio code, the following design was able to be implemented to accomplish parts of the described project.

### II. Design

The deliverables for this project included using the LCD screen of the EFM8 to display data received by the UART pins (running at 115200 baud). The data being received would be keystrokes from a USB keyboard connected to a Raspberry Pi, which would in turn transmit the data to the EFM8. Additionally, one of the pushbuttons on the EFM8 would be used to toggle the output from normal characters to ASCII values in hex. If the screen is filled with characters, the joystick should be able to scroll up through at least 10 lines of history on the display.

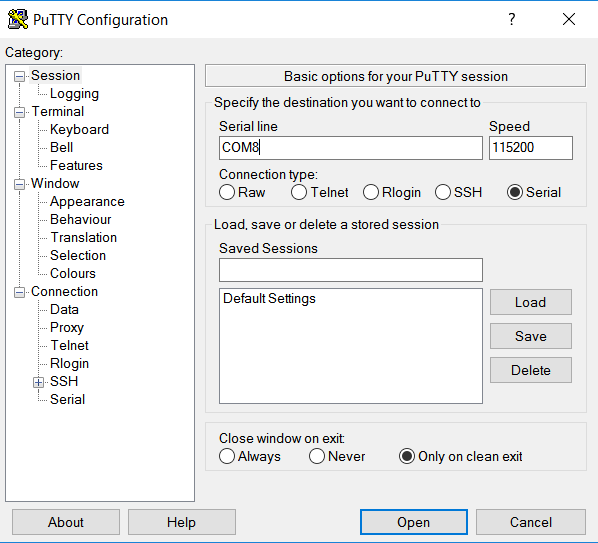
The project was divided into three main parts: installing Raspbian onto the Raspberry Pi and displaying its console to its serial port, displaying the serial data on the EFM8, and writing bare-metal code on the Raspberry Pi to send the keyboard strokes to the serial port.

*A. Part 1 - Setting up the Raspberry Pi*

In order to start working on the project, Raspbian (a Linux-based operating system for Raspberry Pi) had to be installed on a micro-SD card, which was inserted into the Pi. Before being able to access the serial data, the configuration file needed to be edited. By default, the UART transmission and receiver pins will not communicate over a serial port. To fix this, an edit needed to be made to the config.txt file. Specifically, the line “*enable\_uart=1*” was added at the end of the file. After this was done, the Raspberry Pi could communicate over the RX and TX pins of the UART at a baud rate of 115,200. This rate is pinned to the frequency of the Raspberry Pi’s processor.

Once the configuration file was altered, PuTTY was installed to interface with the serial port on the Pi. Using an FTDI cable to connect the transmit and receive pins on the Pi to the USB port of the laptop and running at 115200 baud, the Raspbian console was able to be displayed to the PuTTY terminal.

*Figure 1 - PuTTY terminal*



*B. Part 2 - Display to EFM8*

The next step was to display this data from the serial port onto the EFM8 LCD screen. To accomplish this, C code was written in SimplicityStudio utilizing a function called *drawScreenText* adapted from the Voltmeter example project in SimplicityStudio that takes a pointer to a string, a y position on the screen, and a font size as parameters and in turn prints the string at that position in that size font on the screen. This function required the use of a screen buffer, which stores the values of each pixel on the display. Each time *drawScreenText* was called, the screen buffer would be updated to include that line of text.

The main function of the code, a prompt to enter a character is printed at the top of the LCD. For each iteration of the while loop, a character is retrieved from the UART using the *getChar* function in stdio.h. The character was then stored in an array of characters, *chars*, which was then passed to the *drawScreenText* function, printing *chars* to the LCD. Every time the length of *chars* reached the end of the line on the LCD, *chars* was cleared and the index was reset and the y position was incremented so that the next character would be printed on a new line. *chars* was stored as an external variable to prevent overflow of memory registers that stored local variables in the code.

In order for this code to be executed properly, many of the peripherals had to be initialized properly in the file InitDevice.c. The peripherals used included the UART, timers, pushbuttons, GPIO pins, and the joystick. Additionally, external interrupts had to be enabled, and the watchdog timer disabled. The clock frequencies and baud rates had to be set properly to match the input to the UART. Also, in order to make sure that the UART could be transmitted to by the Pi, the EFM8 board control of the UART had to be disabled as well. This was accomplished by setting pin 2.2 low in our main file.

Although the scrolling, and hex functions were not implemented in the final demonstration, the following describes how these features could be added to the SimplicityStudio code.

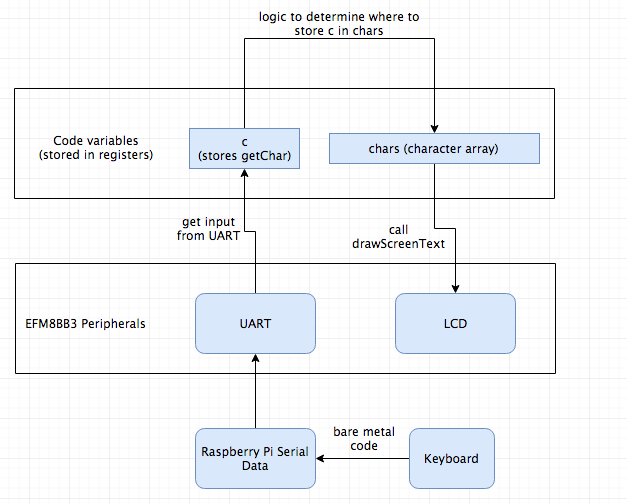
Our implementation of the code placed the characters that were read by the EFM8 into a one dimensional array. When we displayed the characters, we had a for loop loop through the array until the end of the line then loop back to the start of the array to print it. To implement a feature to store 20 lines and scroll we would need to change the way that the array was displayed. To do this we would have the array store 400 characters and create a “first line” variable that would store the multiple of 20 that would be the line. This would count as the first line so that the program knows where, the start was, then there would be another variable “number of scrolls” this would increment every time that the joystick down was pressed and decrement every time the joystick up was pressed. This would be used to display the array offset so as to provide a scrolling functionality. When we ran out of space we would clear the first 10 characters in the next line and set the line after it to line 0 in the line variable. When we would want to print the array out they would simply use a combo of the incremented scroll and the zero position to print correctly.

Implementing hexadecimal characters would require a separate array to store ASCII values, perhaps called *hex*, that would be called similarly to *chars* when a character is received from the UART. Additionally, a function *charToHex* would need to be implemented in order to convert the received character to a hex value. Once the character is passed through *charToHex,* it can then be stored in the *hex* array and printed in the same way.

In order to toggle between character and hexadecimal mode using a push button, the code would require an interrupt service routine to handle an external interrupt from the push button. The interrupt would toggle a boolean value keeping track of the mode. In the main function, this mode would control whether to pass the character through *charToHex,* whether to store in *chars* or *hex,* and whether to pass *chars* or *hex* to *drawScreenText*. This could be easily implemented using an if statement at the beginning of the while loop.

*C. Bare Metal Code*

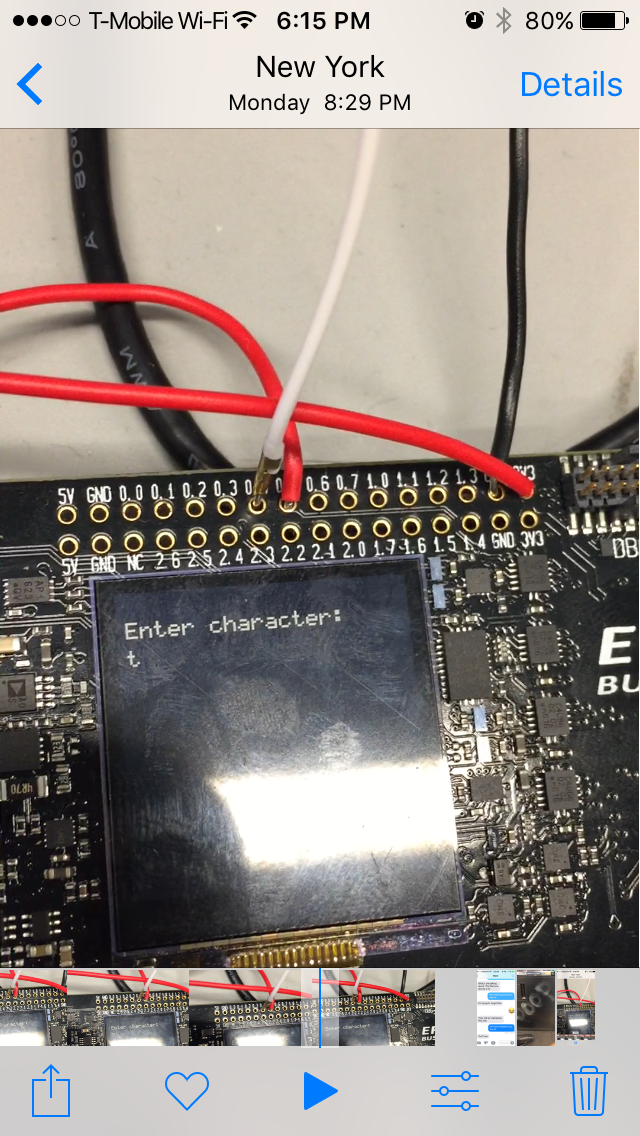
This part of the project was never implemented, however this portion would have been written in the ARM assembly language and loaded onto the Raspberry Pi micro-SD card to run in place of Raspbian, which was previously being used to transmit the serial data. In the actual implementation of the project, keystrokes were transmitted directly from a personal computer, bypassing the need for this bare metal code.



*Figure 2 - Block diagram showing interaction between Raspberry Pi, EFM8, software, and I/O devices.*

### III. Conclusion

In the end, the SimplicityStudio code was able to receive keystroke data from a laptop via PuTTY and display the characters on the EFM8 LCD display, succesfully creating new lines (see Figure 3).



*Figure 3 - EMF8 LCD display. The code displays a prompt, and once characters were entered, they would be printed to the screen. The wires on the ports above the screen connected the EFM8s UART0 RX and TX pins to the USB port on the laptop via FTDI cable.*

Although the hex and joystick portions of the project were not implemented, explanations of their code are detailed above. In this demonstration, the UART on the EFM8 was connected to a personal computer rather than the Raspberry Pi, however interfacing with the Pi would simply require connecting the R/TX pins from the Raspberry Pi serial port to the R/TX pins of the EFM8 UART0. Because the serial data from the Pi was able to be accessed in part 1 of the project, it would be relatively simple to display the console to the LCD display.

The bare metal portion of the project was not implemented either, meaning we were unable to read keystrokes from a USB keyboard to the Raspberry Pi. As explained before, we were able to read the serial data off of the Raspberry Pi, so if a bare metal code had been implemented, it would only require making the right connections to be able to display the keystrokes on the LCD.

Overall, our project demonstrates a working knowledge of the EMF8 LCD display and UART, as well as the Raspberry Pi serial data and operating system. Implementation of all missing aspects of the project are detailed above, and with more time could have been incorporated as well.