Utilizing the UART Peripheral on a EMF8BB3 Microcontroller and Raspberry Pi to Simulate a Text-Based Interface

Joseph Bentivegna and Nikola Janjušević

***Abstract*- The Asynchronous Receiver/Transmitter (UART) is a common microcontroller peripheral capable of serial communication between devices. UART connections are located on the the EFM8BB3 and Raspberry Pi 3 microcontrollers. The Raspberry Pi was first used as the source of data input through the UART to the LCD display on the EFM8BB3. Next, a text-editor interface was added via a computer such that the user could write any text in the terminal and it would show up on the display. Features such as read/write modes, and ASCII to Hexadecimal were implemented as well.**

I. INTRODUCTION

The aim of this project was to experience with the UART peripheral as well as explore means of data transfer between microcontroller devices. The first project design consisted of the Raspberry Pi 3 connected to the EFM8BB3 via the UART TX/RX pins as well as a common ground. When the Raspberry Pi was powered on, a large data stream of startup information streamed through the UART onto the LCD screen of the EFM8BB3 microcontroller. The second setup included a computer connected to the UART pins of the EFM8BB3 via a FTDI cable. The computer acted as a text-editor interface and allowed the user to write directly to the display of the LCD screen.

In both cases, the EFM8BB3 microcontroller was operating on C code that allowed it to consistently take data in from the UART peripheral and neatly display it to the screen. The code additionally makes usage of external and button interrupts, the LCD display, and the ADC peripheral. The user can press the left button to re-enable 'write' mode and can press the right button to toggle ASCII to HEX conversion while in 'read' mode. North/South input commands from the joystick allows for scrolling the text on the screen.

The Raspberry Pi was configured to boot to Raspian on startup which, by default, sends a stream of characters out the UART pins. This was used as an input file stream to test initial UART receiving functionality on the EFM8BB3.

II. DESIGN

A. *Configuration of UARTs and Interfacing*

The microcontroller UARTs are configured communicate at a setting of 115200/8-N-1 (denoting baudrate, data bits, parity, and stop bits respectively). As the device requires only transmission from the Raspberry Pi (RPi) and only receiving from the EFM8BB3, only those channels are required to operate.

The EFM8BB3 UART is configured to receive text input from a command line interface/terminal. In Case 1 of receiving the Linux start-up stream Raspberry Pi, the UART pins of the EFM8BB3 are connected directly to those of the RPi. In Case 2 of simulating the operations of a text-based interface, the EFM8BB3 can either be accessed directly from the RPi Linux Operating System or from a personal computer (PC) through the of use of a serial port to USB cable (in the form of an FTDI cable).

In either operation of Case 2, a command terminal is required to access and write to the EFM8BB3 display. The Raspberry Pi offers a more rudimentary access through the built-in communication through its own UART. A simple "*sudo screen /dev/ttyS0"* command in Bash may suffice to access the dedicated serial port. Access through a PC requires the installation of cable drivers and a search for serial port associated, perhaps even installation of additional software . Both approaches yield the same result: transmission to the EFM8BB3 UART through a command terminal.

On the receiving end of this transmission, the EFM8BB3 stores the captured data in a single byte buffer. This buffer is configured behind the scenes to be recognized as a character accessible to the input stream of the C standard library functions included within standard input output (*stdio.h*). By implementation through this standard library, the *getch()* function effectively accesses the character stored in the UART buffer. This access of the UART data through *getch()* becomes an important constraint in further programming design choices.

B. *EFM8BB3 Peripheral Implementation*

The use of the EFM8BB3 microcontroller as a console output of the RPi start-up as well as text-based interface relies heavily on the embedded peripherals. This includes UART peripheral (as mentioned previously) as well as the embedded LCD, timer peripherals, and the external buttons. The configurations of these peripherals are derived from the stopwatch.

The UART used is one of two available, UART0. This serial peripheral requires the configuration a timer peripheral, timer1, to generate the baudrate of the communications. This timer1 is set up in auto-reload mode to generate receiving interrupt flags. This gives UART0 the requisite ability to do operations at clock cycles asynchronous to the microcontroller CPU.

The LCD configuration is derived entirely from the stopwatch project implementation. Text is drawn row by row using the provided *drawScreenText* function. The use of this function causes limitations to the speed of writing to the LCD, which is arguably matched by the ease of implementation. The consequences of this use are explored in ready of the RPi start-up output in Case 1 and the implementation of scrolling in Case 2.

An embedded joy-stick is implemented using the analog to digital converter (ADC) to convert voltages to directions. This peripheral is implemented via interrupts to provide a scrolling mechanism to the text-interface.

The external button interrupts offer the fluidity of the user-interface by supplying precise response times. This is achieved through hardware which is able to operate separate to the clocking of the CPU. However, as a result of this isolation the diminished stack space requires the instantiation of global binary valued variables to act as “pseudo functions”. Through these pseudo functions communication takes place between interrupts and the main EFM8BB3 code.

As a result, the main loop consists of checking the values (‘on’ or ‘off’) of these binary valued globals. This takes the form of three functions: *redrawScreen, insertchar,* and *scroll_check. redrawScreen* serves to check if any mode updates or scrolling must be drawn to the display. The *insertchar* function serves to allow for the insertion and drawing of characters in the current row of the buffer only if the device is in write mode. The *scroll_check* function updates buffer window values based upon joystick direction if the device is in read mode.

C. Program Motivated Design Choices

The serial communication device possesses two fundamental design choices on the display end present from the standard library set up of the EFM8BB3 UART and the choice to favor Case 2 operation over Case 1 operation of the device.

The implementation of UART0 using the C standard library causes the program to latch onto the *getchar* function while waiting for user input. From this, it becomes impossible to perform another operation without first transmitting a new character to pass through the *getchar* hangup. To circumvent this issue, the device implements separate write and read modes. Write mode is enabled by default and left by entering the escape key on the keyboard. When in read mode, the *getchar* function is no longer accessible to the main loop and thus further operations are possible. Most notably, this allows for up and down scrolling using joystick. A buffer of 10 lines extra to those available on the EFM8BB3 screen are used to store ‘scroll-able’ information. As a bonus feature, this implementation allows for immediate translation upon switching between ASCII and HEX modes when in read mode.

The text-interface lends itself to live typing character by character for more fluid user interface. To achieve this, rows must be redrawn upon the insertion of every new character. This operation is slow and taxing, however unnoticable in Case 2.

Case 1 has the RPi sending a start-up strem quite quickly and thus requires a quick method of saving and display this information. The proper way to implement a Case 1 file stream to the LCD would be to store each row in a character buffer and only draw to the display when a newline/row is reached. This would cut down on CPU operations and allow for proper receival of the stream.

The favoring of Case 2 in this device is readily seen in the performance of Case 1, where several characters are lost due to the greater set of operations happening each cycle. This is a necessary consequence of favoring the live-typing interface.

C. User Interface Motivated Design Choices

Several design choices of the text-editor interface of Case 2 come down to a better user experience. This choices include the screen size, the simultaneous use of ASCII and HEX buffers, text-wrapping, and implementation of backspace functionality.

The screen size of the EFM8BB3 embedded display allows for a relatively large text row and thus a number of characters too large to store in the standard 8051 bank registers. Instead, the ASCII and HEX buffers (2D arrays of characters) are stored in external memory beside program memory. A drawback of this larger memory is the slower access time, easily recognisable in the slow draw time of scrolling.

These ASCII and HEX character buffers have characters inserted simultaneously regardless of which mode the user is currently in. This is to facilitate the switch between ASCII and HEX modes so that previous text need not be translated from one buffer to another. Upon pressing the button to switch modes, the user sees an instantaneous translation from one character set to another.

To build this device as a tool to easily translate between ASCII and hexadecimal, it is necessary for all characters currently on the screen to remain on the screen (in the translated value) after transition of modes. It is for this reason that text wrapping is implemented into the device interface. Each hexadecimal is printed as 3 characters, and thus a line of ASCII easily fills the next lines in HEX mode. Text wrapping ensures nothing is lost in translation. In addition, it creates a smooth user interface where no user input is required to reach the next line (as opposed to an interface where entering a new line character is necessary).

This last user interface design choice is one of pure convenience: the backspace. This feature comes easily from recognizing the ASCII value for the backspace key, jumping between rows/characters, and deleting entries accordingly. Entries from the ASCII and HEX buffer are deleted together to maintain a feel of translation about the device.

The Raspberry Pi 3 microcontroller acts as the sole source of data input into the EFM8BB3 in case 1 of the project. The Raspberry Pi was set up with the Raspian operating system, which, by default, sends boot information through its transmit UART pin. When connected with the UART pins of the EFM8BB3 microcontroller that is running the code, the input text data will appear on the LCD display. Although the Raspberry Pi was not utilized for much more than that, it is important to note that the

Raspberry Pi can act in the same way as an FTDI cable in that it can be the interface between the computer (which has no UART connection) and the EFM8BB3.

III. CONCLUSION

The project largely rested on the C code that allowed coexistence of multiple interrupts with other, user based features. Design considerations within both the code and the user interface allowed for organized and customizable text matching between the computer and display. The inclusion of the computer terminal in the project added another dimension of software consideration regarding how to best code for the user.

Completing an exploration of the UART peripheral has demonstrated the principles behind hardware device information transfer. Any stream of data (granted size limitations) can be sent and received via the UART allowing for similar methodology of transmission/receiving. In the future, considerations will be made about how to increase speed and efficiency of text input in the project.