ECEN 5813 (F'18)

Project - 2

Project Report

Project Details	*Description*
Group Members	**Rushi James Macwan and Poorn Mehta**
Date	**27th November 2018**
Project	**PES Project-2 (Circular buffer, UART and interrupts)**
Class Name	**Principles of Embedded Software ECEN 5813 (Fall-2018)**
GITHUB Repository Link	*https://github.com/MacRush7/Project_1_R-P/*
GITHUB Unit Test File Link	*https://github.com/MacRush7/Project_1_R-P/tree/master/CUnit_Testing/*

[PROJECT REPORT]

University of Colorado Boulder

Created solely by: Poorn Mehta & Rushi James Macwan

Table of Contents

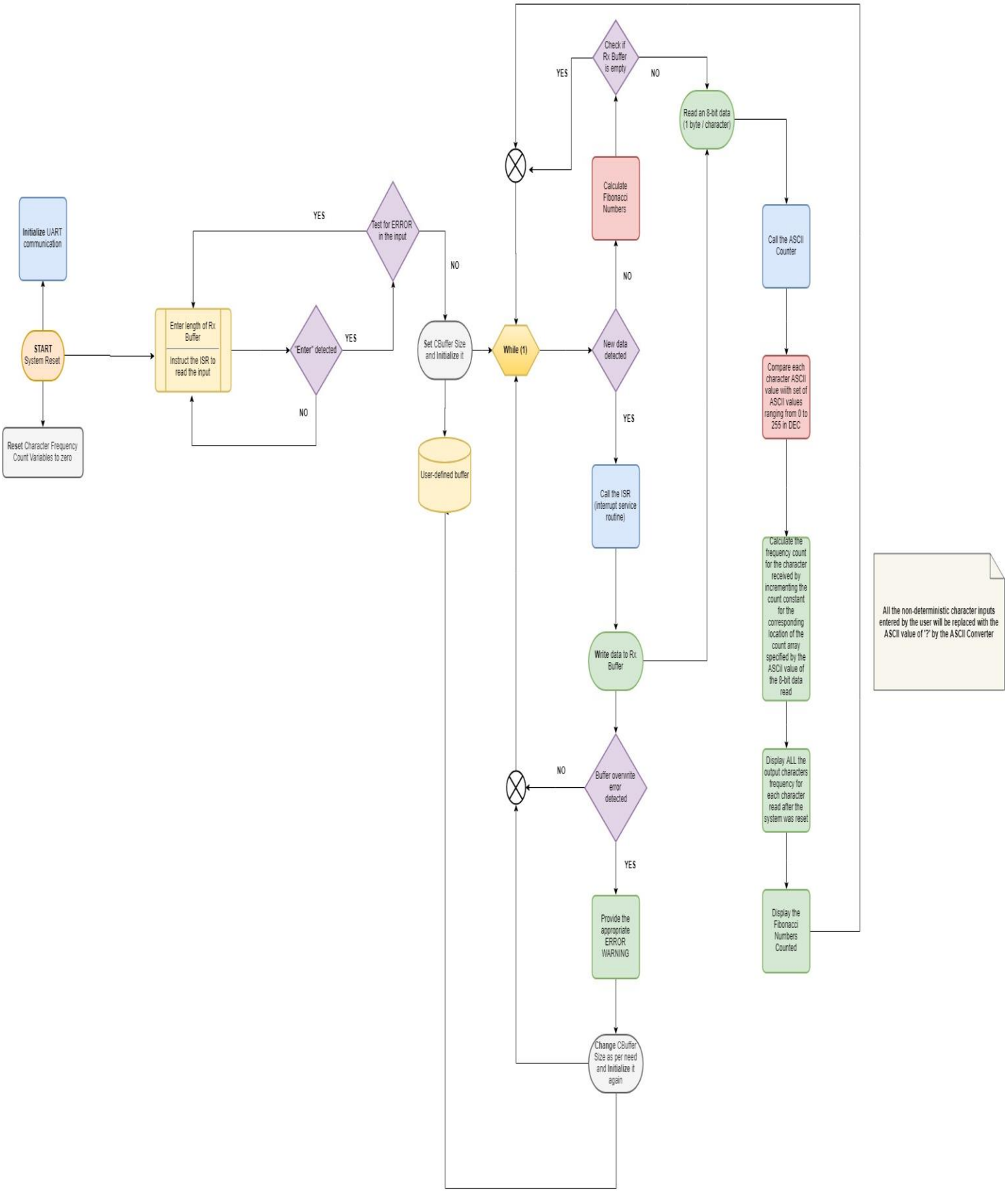
Block Diagram and Architecture Description:

In this Project, we have been asked to work on five different parts which when added together shows a potential evidence of successful use of serial communication to calculate the frequency of characters corresponding to the inputs that the user makes.

To meet this objective, we have been using a circular buffer that is connected with the UART network which acts based on the user's inputs. So, a quick explanation of the parts that this project covers is as under:

1. Circular Buffers
2. UART Device Driver
3. An application
4. Unit Testing Framework

Block Diagram – Can be accessed by clicking **here**. The draw.io version of the same can be accessed by clicking **here**. A screenshot has also been added in this report.


Initialize UART communication
START System Reset
Reset Character Frequency Count Variables to zero
Enter length of Rx Buffer
Instruct the ISR to read the input
'Enter' detected
YES
NO
Test for ERROR in the input
YES
NO
Set CBuffer Size and Initialize it
User-defined buffer
While (1)
New data detected
NO
YES
Call the ISR (interrupt service routine)
Write data to Rx Buffer
Buffer overwrite error detected
NO
YES
Provide the appropriate ERROR WARNING
Change CBuffer Size as per need and Initialize it again
Check if Rx Buffer is empty
YES
NO
Calculate Fibonacci Numbers
Read an 8-bit data (1 byte / character)
Call the ASCII Counter
Compare each character ASCII value with set of ASCII values ranging from 0 to 255 in DEC
Calculate the frequency count for the character received by incrementing the count constant for the corresponding location of the count array specified by the ASCII value of the 8-bit data read
Display ALL the output characters frequency for each character read after the system was reset
Display the Fibonacci Numbers Counted
All the non-deterministic character inputs entered by the user will be replaced with the ASCII value of '?' by the ASCII Converter

Architecture Description:

Based on the block diagram that can be accessed by clicking the above link, the project flow can be explained as under:

1. **System Reset**

   This is the primary point in the implementation of this project from where the execution of the code begins.

2. **UART Initialization**

   The UART is initialized every time after the system is reset. This enables serial communication of the target with the host through which the user actually transmits the inputs.

3. **Resetting the character frequency count variables**

   The character frequency count variables are initialized and set to zero value every time a system reset is performed. This implies that every time a reset is performed, the application will start calculating the character frequency counts from zero; ignoring the counts previously reported.

4. **Specifying the buffer length**

   The user is then requested to specify a length in terms of characters for the circular buffer that will store the user entered data.

5. **Interrupt Service Routine (ISR) is requested to start reading the inputs from the user for the data to be reported and stored to the buffer**

   As can be read from the title, the ISR is requested to start reading the input data entered by the user once after the user specifies the buffer length and it has been initialized successfully.

6. **Test for error in the input given by the user for buffer initialization**

   The buffer length is further tested if it falls within the specified limits that the processor can handle. If no error is detected, the user is allowed to enter data to the system. Otherwise, the user is expected to enter a valid buffer length.

7. **Buffer initialization**

The buffer is initialized and is ready for use once the user provides an acceptable buffer length.

8. WHILE(1) loop entry

From this point onwards, the code enters the WHILE(1) loop and the ISR starts looking out for any data that the user enters through the UART channel.

If data is available: The ISR will start writing the data to the assigned and initialized R_x circular buffer. It will also check if the buffer is full based on the index flag status and it will produce an "overwriting" warning if the buffer is "overwritten". Moreover, if the "overwriting" takes place, the buffer length will be appropriately resized to contain the characters inevitably. This implies that the buffer will be re-initialized with a new length that can hold the characters that are "overwritten".

If data is unavailable: The CPU will start calculating Fibonacci Numbers and will simultaneously check the buffer if it is empty. As long as new data is unavailable or the buffer is empty, the execution will keep on calculating the Fibonacci Numbers and will stop doing it once the condition is met.

As soon as data is available, the ISR will start writing the data to the assigned and initialized R_x circular buffer. Once the buffer has some data in it written by the ISR, the buffer empty test will fail and therefore the execution will start reading bytes from the data stored in the buffer. The ASCII counter function will be called during this stage and it will start comparing each character received with the range of ASCII values in Decimal: 0 to 255. Once a match is obtained, the ASCII counter will increment a location in the count array corresponding to the match value ranging from 0 to 255. Thus, the character frequency count is performed and the report is printed after every new character is received since this code is written inside the WHILE(1) loop. Also, with this, the calculated Fibonacci numbers will also be displayed.

Note: In the process of calculating the frequency of characters received from the user, it is possible that some non-deterministic characters are received by the UART driver and are stored in the buffer. The ASCII counter function in turn will replace the non-deterministic characters with the character: '?' and its ASCII value. Thus, the system is susceptible to an error in the calculation of the character '?' if there are any characters entered by the user that are non-deterministic.

Some important functions/macros used:

The important functions/macros used for operating on the circular buffers are as follows:

Byte CBuffer_Assign(Byte CBuffer_ID)
Byte CBuffer_Init(void)
Byte CBuffer_Byte_Write(Byte CBuffer_ID, Byte data)
Byte CBuffer_Byte_Read(Byte CBuffer_ID, Byte *address)
void CBuffer_Operation(Byte CBuffer_ID, Byte type, Byte data, Byte *address)
DWord CBuffer_Elements(Byte CBuffer_ID)
Byte CBuffer_Resize(Byte CBuffer_ID)

In a similar fashion, the important functions/macros corresponding to the operation of the UART are as follows:

Enable_UART0_Rx_Function()
Enable_UART0_Tx_Function()
Enable_UART0_Tx()
Enable_UART0_Rx()
Disable_UART0_Tx()
Disable_UART0_Rx()
Enable_Rx_Interrupt()
Enable_TxE_Interrupt()
Disable_TxE_Interrupt()
UART0_Wait_for_Tx_Data_Register()
UART0_Wait_for_Rx_Data_Register()
UART0_Rx_Interrupt()
UART0_TxE_Interrupt()

In addition to the above mentioned details, the ASCII counter block will include a few more functions/macros as below:

ASCII_Counter_Init(void)
ASCII_Counter(void)

Report Questions:

1. Is your implementation thread safe? Why or why not?

   The implementation can be considered thread safe in the absence of the interrupt routine. Thus, when the implementation is performed while the interrupts are disabled, the variables are localized to their respective threads and these variables retain their values across the thread code boundaries. Thus, the data fed to the circular buffer in this case will be a thread safe implementation given that the interrupts are enabled after the buffer operation is complete.

   Courtesy: {here}

2. What potential issues exist if the same buffer is used by both interrupt and non-interrupt code? How can these issues be addressed?

   One of the potential issues that exist if the same buffer is used by both codes has to deal with the 'race' condition or in other words – a concurrency bug. This will only happen if both code use the same buffer operation – that is if both interrupt and non-interrupt code are allowed to read or write to the buffer which can cause some issues.

   To solve this, we have provided the interrupt service routine with the ability to only write to the buffer. On the other hand, the non-interrupt code has been provided the ability to only read from the buffer.

3. How could you test these issues?

   The efficient use of Unit Testing modules in developing reliable test conditions for removing the 'race' condition or otherwise a concurrency bug can help test the issues. If the tests run successfully on the code then it can be said that the code is proof from such bugs.

   We have tried to touch some basic outlines of this test framework in our project.

4. For each implementation, what is the CPU doing when there are no characters waiting to be echoed? What is the behavior of the GPIO toggle in the non-blocking implementation?

   In the absence of characters that are waiting to be echoed, the CPU will rest and wait while a character is received from the UART. This will be true for the blocking mode since the interrupts are not used in this case but polling is used.

   However, in case of the non-blocking mode, the CPU will continuously toggle a GPIO pin when it is not busy echoing the characters. This is true since in this case we are using interrupts as opposed to the polling that is used in the blocking mode.

5. For each implementation trace the sequence of events that occur by listing, in order, the functions called from the point that a character sent to the FRDM board has been received until the point where the echoed character has been sent.

   To understand the processes better, the architecture description explained in the above section in this report will throw more light on the underlying implementation.

   As for the functions/macros called from the point when a character is sent to the FRDM board has been received until the point where the echoed character has been sent, the below information will explain more:

   Blocking mode (polling):

   The polling mode will contain the following basic blocks including the subsidiary ones introduced in the code:

   1. Custom_UART0_Init()
   2. UART0_Wait_for_Rx_Data_Register()
   3. Custom_UART0_Rx_Byte(&test)
   4. UART0_Wait_for_Tx_Data_Register()
   5. Custom_UART0_Tx_Byte(test)

   Non-Blocking mode (interrupts):

   The non-blocking mode will actually use interrupts and will toggle an LED if an interrupt occurs. Once again, it will contain the following basic blocks *in addition to the ones in the blocking mode* including the others present in the code:

   1. Enable_Rx_Interrupt()
   2. NVIC_EnableIRQ(UART0_IRQn)

   These two blocks ***are in addition to the general UART initialization blocks*** as mentioned above in the blocking mode case which are as under:

   - Custom_UART0_Init()
   - UART0_Wait_for_Rx_Data_Register()
   - Custom_UART0_Rx_Byte(&test)
   - UART0_Wait_for_Tx_Data_Register()
   - Custom_UART0_Tx_Byte(test)

   Furthermore, the code in the non-blocking mode will also include Interrupt Service Routine (ISR) blocks and will involve LED toggling as below:

   1. UART0_IRQHandler(void)
   2. UART0_Rx_Data(&UART0_Byte) – Writing the data into the UART R_x circular buffer

At the same time, the T_x interrupt flag will come into picture while it is enabled and is notifying if the T_x transmission is complete through the interrupt flag. If it so, the IRQ handler will come into play if the UART is required to transmit the buffer data.

The LED toggling will involve the following three macros for the LEDs with different colours providing three different test cases of LED toggling.

1. BOARD_GPIO_LED_BLUE
2. BOARD_GPIO_LED_RED
3. BOARD_GPIO_LED_GREEN

6. Comment on the interface presented to the main() application code for blocking vs. non-blocking variation. Which variation is easier to code to?

   The non-blocking code will use interrupts to actually allow the CPU to perform other operations freely without having to wait for a condition to be met. This will allow us to efficiently use the CPU – such that it can enter the sleep mode or it can do nothing and therefore it can stop wasting energy in these cases. Moreover, whenever a certain situation arises that is important for the CPU to deviate from the normal code execution, the interrupts will take over the lead.

   As opposed to the non-blocking code, the blocking code will use the easy method of polling an internal status flag for report reception completion. It is set whenever a new byte (character) is available in the buffer that can be read. This implies that the CPU will be idle and will simply wait until the reception is completed.

   Given the coding architecture of the modes above, the blocking code is easier to code to. The reason is that the code in the blocking mode will only be containing a simple polling feature and therefore the code will be simple. However, for the non-blocking mode, it demands writing interrupt service routines (ISR) that deal with the CPU which is tedious and not simple most of the time. Also, the interrupt service routines should look out for concurrency bugs and must protect shared resources so that the operation remains free from any flaw.

7. What is the CPU doing after the last character has been received and while the report is being printed?

   The CPU will perform calculation of Fibonacci numbers until the last character has been received and while the report is being printed.

8. Baud rate aside, what limits the rate at which the application can process incoming characters? What happens when characters come in more quickly than they can be processed?

Aside baud rate, the rate at which the application processes the incoming characters is limited by the ISR handling capacity or in other words the rate at which the code is capable of writing data to the buffer. This is in turn dependent on the processor speed that the development board uses.

When characters come in more quickly than they can be processed, the character frequency calculation output might show a value that is lesser than the actual frequency of characters that the user enters – this means there will be an error in the character frequency output. Moreover, it will also tamper the report that is echoed back since some characters might not be reflected back in the echoed response.

9. How does the size of the circular buffer affect report output behavior (especially during an onslaught)? What is an appropriate buffer size to use for this application? Why?

In our case, the user is first asked to specify the length of the buffer that is going to be used for reading/writing the report (the characters that the user enters). If the size is lesser than the expected report size, then the echoed response will give out a misprint since some characters might not be reflected back. In simple words, this will create errors.

We have tried to incorporate a feature in the code where the buffer will adapt to the size of the report entered into the system and therefore there will not be any issue with data entry and buffer size.