Colin Young
Lab 2

In 1837 the great Samuel Morse, along with his friends Joseph Henry and Alfred Vail developed a method of transmitting natural language across great distances by using only electrical signals. In the year 2024 I, Colin Young, along with a team of expert engineers (Aaron Goethel) are going to reinvent the wheel. We will be composing a program that takes ASCII input and translates that input into Morse code, making a little LED blink on a NUCLEO board 3 feet from our computer. To achieve this, we will be using the newest (to me) piece of technology on the market today, the Fanateek FDI cable.

Requirements:

1. Set up UART Parameters

2. Set up Tera Term Parameters

3. Wire the NUCLEO board to receive input via the FDI cable

4. Design and implement an algorithm that takes user input, translates it into morse code, then blinks the blue led on pin B7 accordingly.

Verification:

1. We will be utilizing UART3 for this program. We will use the standard baud rate of 115200, an 8 bit word length, no parity bit, and 1 stop bit. For further details please refer to technical details part A.

2. Tera term must match our UART3 parameters. Confirm the Baud rate is set to 115200 8N1 and we using the COM connected to the USB port with our FDI cable. For further details please refer to technical details part A.

3. To wire up the NUCLEO board properly connect each colored wire of the FDI cable to the corresponding pin:
    a. RED +5 - CN8 Pin 9
    b. White TXD - CN10 PD9 USART_A_RX
    c. Green RXD - CN10 PD8 USART_A_TX
    d. Black - GND - CN8 Pin 11

4. The algorithm will work properly if it can take in user inputted text data via the tera terminal, and blink the blue led located on pin B7in accordance to the Morse code standards. This will take advantage of the UART interrupt described in technical details part B. The full algorithm description can be found in technical details part C.

Technical Details:

A)

In order to get the UART connection running smoothly, we must first take a look at what the HAL_UART_Transmit function is doing. Below is a list of the parameters the function takes, and an explanation of each one.

Function and parameters:

HAL_UART_Transmit(UART_HandleTypeDef *huart, uint8_t *pData, uint16_t Size, uint32_t Timeout)

UART_HandleTYPEDEF *huart :

This argument uses the *huart pointer to point to an UART device handle. In our lab we use hlpuart1, and huart3.

Uint8_t *pData:

*pData points to an array that we want transmitted, In the first part of the lab that is the tx_buffer variable.

Uint16_t Size:

Size represents the size of the array, in bytes, sent. Letting the handle know when the array has been transmitted.

Uint32_t Timeout:

The timeout variable exits the function and returns a timeout error if the operation exceeds the time limit, preventing the function from hanging.

Now that the transmit function is understood, we can set up tera term to listen for the board on the corresponding com port and simply send a message via HAL_UART_Transmit in our main while loop to know we have everything set up correctly. Once we have received the transmit in tera term and know we have everything set up properly, we can remove the transmit function, and move on to implementing our algorithm.

B)

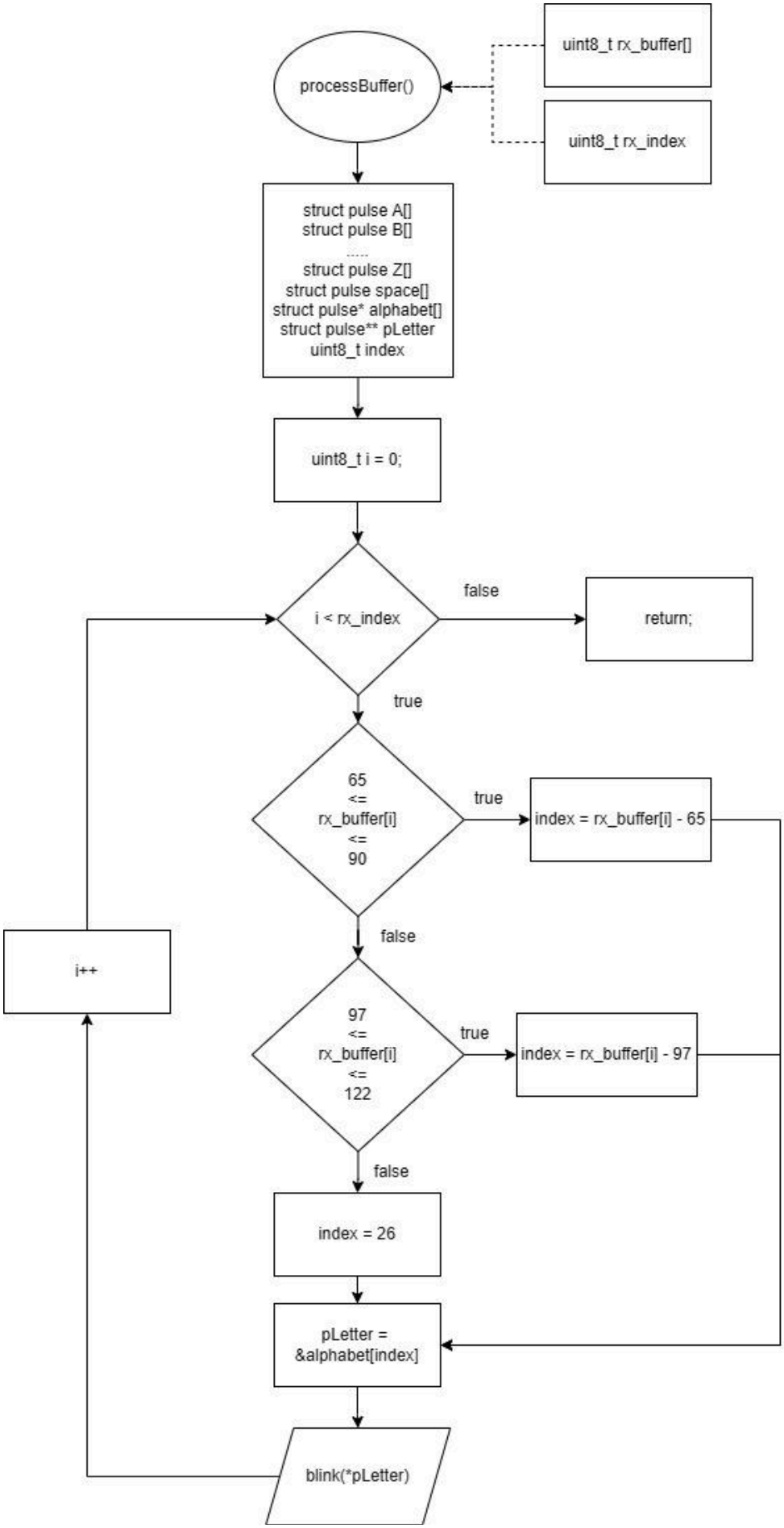
User input will be received through the USART_A_RX on pin PD9. In order to read this information we will be utilizing the HAL_UART_RxCpltCallback() function. This function, seen on

line 643 of the attached code, waits for an interrupt on usart3, then stores the incoming data onto an rx_buffer.
If the function reads in the value of 13, which represents a carriage return in ASCII, we proceed to process the data stored on the rx_buffer, and then reset the index to 0 so the buffer will be cleared out on the next interrupt.

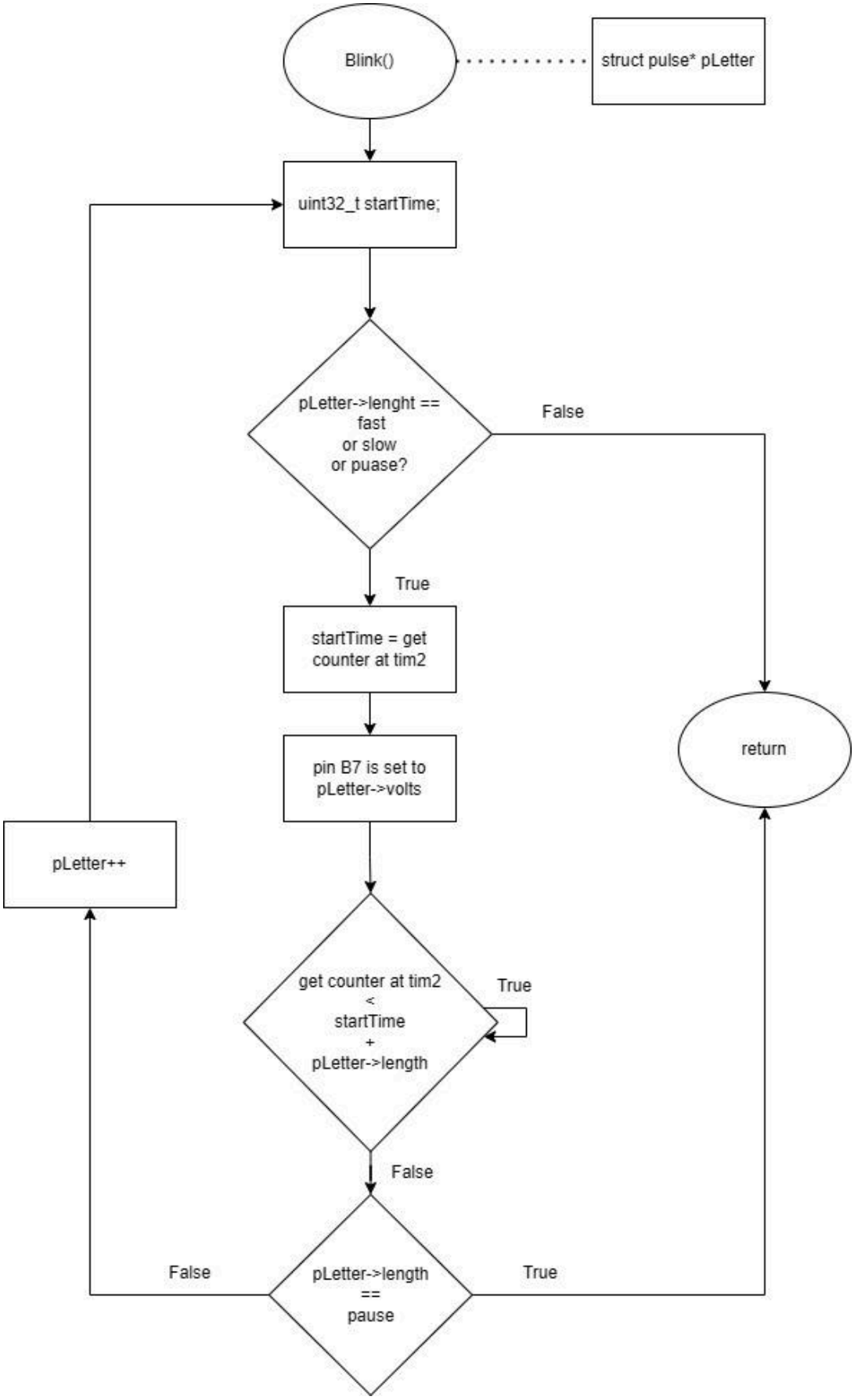
C)

The main algorithm of this project is designed to take the rx_buffer and transform that into morse code by blinking the LED assigned to pin B7. In order to represent morse code in the project we must first break it down to its base components and make rules around those components. At its base, Morse code is an organized series of pulses. The pulses themselves have a length, and a voltage. The length represents how long the pulse runs for, and the voltage is whether the pulse is high, or low. In morse code there are four types of pulses, a dash, a dot, a pause, and a break. In order to standardize the pulses, we will set the dot and pause length to 0.25 seconds, the dash to .75 seconds, and the break to 1 second. The dot and dashes will have a high voltage, while the pause and break will have low voltage. Each letter in the English alphabet can then be represented by a unique series of pulses ending with a break. The letter A for example can be represented: dot, pause, dash, break. To translate this into c code, we will create a structure called pulse, which will have the variables length, and volts. We can create four instances of this structure representing the dot, dash, pause, and break. A letter can then be represented as an array of pulses and the full alphabet can in turn be represented by an array of letters.

With the rules set on how we will be handling morse code, we can proceed to process the rx_buffer. Implementing the HAL_UART_RxCpltCallback() function described above, we wait until the carriage return is read before sending the rx_index and rx_buffer to the processBuffer function, the diagram of which is seen below. Inside this function we set up all of our letters as well as the alphabet array. We then iterate through the rx_buffer array, checking each index to see if it is equal to A-Z represented by ascii code 65-90, or a-z, represented by 97-122. If the value falls outside of that range we default it to a space. The letter is then sent to the blink function which processes the letter and blinks the LED accordingly.

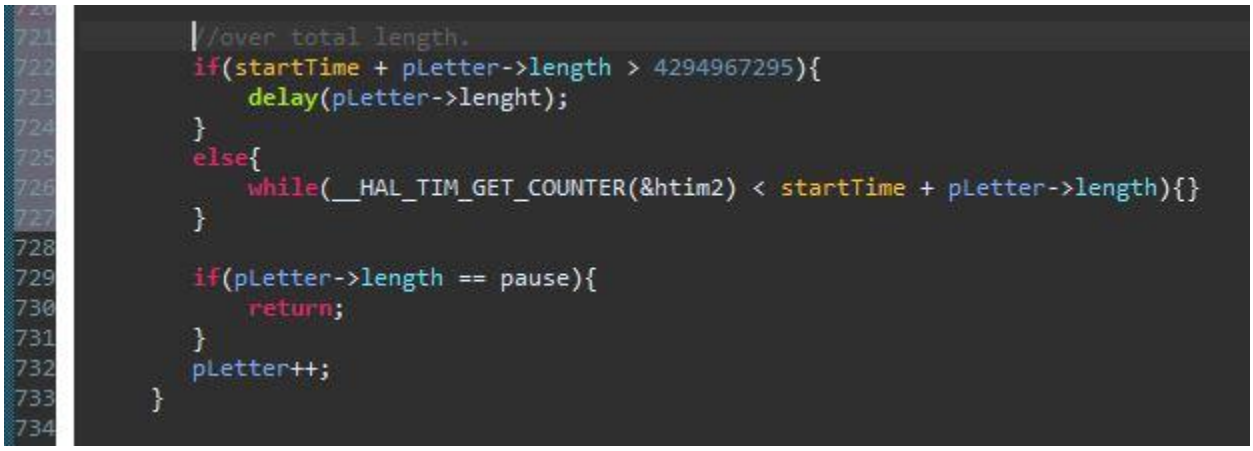

uint8_t rx_buffer[]
processBuffer()
uint8_t rx_index
struct pulse A[]
struct pulse B[]
.....
struct pulse Z[]
struct pulse space[]
struct pulse* alphabet[]
struct pulse** pLetter
uint8_t index
uint8_t i = 0;
i < rx_index
false
return;
true
65
<=
rx_buffer[i]
<=
90
true
index = rx_buffer[i] - 65
false
i++
97
<=
rx_buffer[i]
<=
122
true
index = rx_buffer[i] - 97
false
index = 26
pLetter =
&alphabet[index]
blink(*pLetter)

The blink function is relatively simple. It takes a pointer to a letter, and iterates through the pulses until it processes the break pulse. For each pulse, it sets the LED to the corresponding voltage, and then waits for the correct length of time, before moving on to the next pulse in the letter. A flow chart of the function is listed below.


Blink()
struct pulse* pLetter
uint32_t startTime;
pLetter->lenght ==
fast
or slow
or puase?
False
True
startTime = get
counter at tim2
pin B7 is set to
pLetter->volts
return
pLetter++
get counter at tim2
<
startTime
+
pLetter->length
True
False
pLetter->length
==
pause
False
True

Conclusion:

During the final debugging phase we noticed some issues with the code. While the code ran smoothly 90% of the time, every once in a while the LED would not blink properly. While we ran out of time to test fully in the lab, I believe the issue could stem from the time counter. Our pulses have a length of 25,75, and 100 million, and our timer has a max size of 2^32 or just under 4.3 billion. With each signal sending multiple pulses it is possible that we send a pulse at the upper end of the counter, causing it to hang as it can never exceed the startTime + the pulse length. This bug will be the main focus of our next development cycle. In the meantime as a soft fix, please try appending the following snippet of code to the blink function. Our engineers would have sent it with the main file but they are afraid it might break everything and lack the Fanateek FDI cable to do the testing at their homes. We could also forgo the timer all together and simply implement the delay function for each pause to remove the bug but that isn't as fun.



```
        //over total length.
        if(startTime + pLetter->length > 4294967295){
            delay(pLetter->lenght);
        }
        else{
            while(__HAL_TIM_GET_COUNTER(&htim2) < startTime + pLetter->length){}
        }

        if(pLetter->length == pause){
            return;
        }
        pLetter++;
    }
```