EEET2096 – Embedded System Design and Implementation

Laboratory 3 – Serial Communication

Conor Christensen – s3815282

Wednesday 1430

Introduction

This laboratory will introduce programming an ARM Cortex-M4 CPU using C language, implement a Universal Asynchronous Receiver/Transmitter (UART), and use timer registers for real-time delays. Using the IDE Keil µVision, the students will program using C, a low level sequentially compiling language that can be represented with its ARM assembler equivalent within the IDE. Using a template with imported register address libraries, the students will initialise GPIO registers similar to the second laboratory along with additional GPIO for the UART and other internal timer registers. C is used because it reduces the complexity in the instructions written to allow for larger scale programs to be run.

The laboratory instructs the students to program the Cortex-M4 to process an input received on the UART and to control the fade-in/fade-out time of the LEDS on the development board. This will consist of two main components in the program: the fading routine and the UART receiving routine.

The LED fading routine will require the implementation of two internal timers, TIM6 and TIM7. These are basic timers that take a count value as input (among other initialisation values) and set a flag once the count has been met. The rate at which the timer counts is dependant on a clock signal on the APB1 bus which on the development boards will run at 42MHz. The timers will be used to control the fade.

The brightness of an LED is mainly dependant on the current supplied to it given the threshold voltage is exceeded. The processor cannot vary the current through the LEDS further than the binary on/off states, so therefore we must implement a different method to control their brightness. Pulse Width Modulation (PWM) can be used to control the output light intensity of an LED by running the LED on a square wave voltage cycle. Due to persistence of vision in humans we are unable to quickly detect oscillations in light sources and instead perceive the light to be brighter or dimmer depending on the ratio of on time to off time in the cycle. Using the concept of duty cycle, the percentage of the total period time a square wave is high, we can control the brightness of an LED. If the LED is flashing at a constant rate in Hz, however, has a small duty cycle (effective 'on' time) then it will seem dimmer than if it has a large duty cycle. Using the internal basic timer, we can adjust the count value provided to influence how long an LED will stay on for each cycle and therefore change the effective brightness of the LED. Given the LED will need to fade in and out the timer will have to adjust these count values in a loop to slowly increase the duty cycle to 100% for a fade-in, and slowly decrease the duty cycle to 0% for a fade out.

According to the laboratory guide the fade-in/fade-out can take as long as ten seconds and a short as one second depending on the UART input, with one second increments in the timing. The changes in the fade time will be implemented by loading the tiers with different parameters to handle the adjustment. This can be done through the timer count value (where the timer counts to before signalling an output), or through the timer pre-scale. The pre-scale in the timer is an effective stepper from the input clock. Using a register 16-bits wide, the pre-scaler can reduce the input clock signal to the counter by a factor of up to 65536 depending on the value loaded into the register. For example, if we wished to halve the clock rate, we could load in the value 0x0002 which would require two clock ticks for every counter tick, giving a count rate of 21MHz.

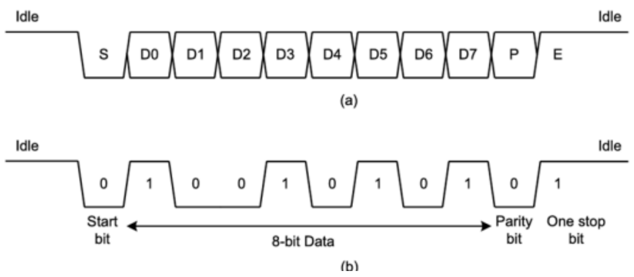


Figure 1. UART transmission bits

The UART is an asynchronous transmission and receiver port that communicated with 11bit transfers across either a receiver (RX) wire or a transfer (TX) wire. As seen in Figure 1, the UART communication is comprised of a start bit, a parity bit, a stop bit, and eight data transfer bits. The parity bit is used to check for a corrupted transmission on the receiver side, and the start and stop bits are used to signal the bookends of the transmission. UART communication is asynchronous, meaning that the transmitting device and receiving device to not operate on a shared clock, and instead the data transfer rate is dependant on how quickly the receiver processes the data. The eight data bits received are loaded into a data register, and only when that register is cleared will the transmitter send another communication. The UART for this laboratory will only require the receive functionality as the communication does not require and acknowledgement or ACK elements.

Using the UART discussed, the students will have to configure the Cortex-M4 for UART communication using the GPIO and receive communications of three characters at a time. These characters will be ASCII characters encoded in eight-bit binary values which is the width of the data transfer in UART. The students will be required to write a program that will process the signal, check for valid values, and then use the values received to influence the fade-in/fade-out rate of the LED according to Table 1 below.

Table 1. LED fade times

Serial Value	Fade-In / Out Time (seconds)	Serial Value	Fade-In / Out Time (seconds)
100	1	040	7
090	2	030	8
080	3	020	9
070	4	010	10
060	5	000	All LEDs Off (Disabled)
050	6	-	-

Results and Discussion

Initially, the laboratory guide asks for the students to form a flow chart which will dictate the program flow. After reading the task thoroughly the chart in Figure 2 was produced below.

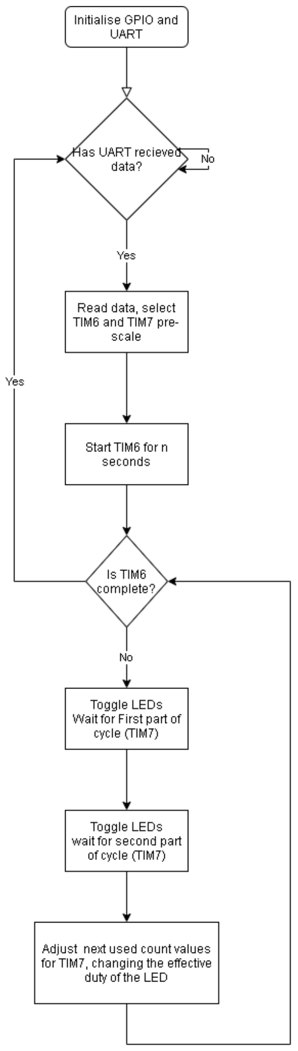

Initialise GPIO and UART
Has UART recieved data?
No
Yes
Read data, select TIM6 and TIM7 pre-scale
Yes
Start TIM6 for n seconds
Is TIM6 complete?
No
Toggle LEDs Wait for First part of cycle (TIM7)
Toggle LEDs wait for second part of cycle (TIM7)
Adjust next used count values for TIM7, changing the effective duty of the LED

Figure 2. Flow chart design.

The rudimentary design in Figure 2 covers the basic elements of the program, and using previous coding knowledge it is understood that with 'if' statements and 'while' loops the conditionality and flow of this diagram should be able to be replicated in the code.

Using reference to the lecture notes as a guide it was understood that the GPIO and UART registers would need to be enabled and configured correctly for this program. To begin, two methods in C were written named configureLEDS() and enableTIM6TIM7() which can be seen in Appendix A. These methods enabled the clock register for the respective GPIO and Timers, and configured the GPIO for the LEDs in GPIOA, GPIOB, and GPIOF for medium speed, output, and to initialise the LEDs as off.

The UART was next to be configured, however would require some calculation regarding the project specifications. The laboratory guide specifies that the UART must be configured to operate with a baud rate of 9.6kBps, 8 data pits, no parity and one stop bit. The baud rate register (BRR) is only 16 bits wide, therefore specifying a baud rate must be done using a mantissa (12-bit) and a fraction (4-bit). The UART will operate using OVER16 enabled, which is a mode where the data is oversampled taking 16 samples for the 8 incoming data bits. The STM32F439 USART-DIV value can be calculated using the following equation:

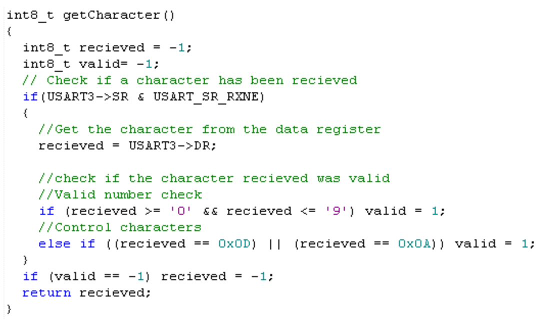
$$USARTDIV = \frac{f_{ck}}{8 * (2 - OVER(8)?) * baudrate}$$

In this case given OVER8 is disabled (OVER16 enabled), the clock frequency is 42MHz, and the baud rate is 9600, we get a value as follows.

$$USARTDIV = \frac{42 * 10^{\wedge}6}{8 * 2 * 9600} = 273.4375$$

The value 273 can be converted straight away into hexadecimal 0x111 and loaded into the mantissa register. The 0.473 value is multiplied by 16 (given the four-bit fraction register) and rounded to the nearest integer, converted to hexadecimal (0x07), and loaded into the fraction register. Given the mantissa and fraction are not exact due to rounding there is potential for errors to occur, however this is reduced with the oversampling. The UART is then configured for all other requirements specified. The UART will communicate to the STM32F439 using the GPIOB pins 10 and 11. These pins were configured according to the technical reference manual such that their mode was set to Alternate Function (AF). Their alternate function register was then loaded with 0x07 to set the alternate function 7, implementing the UART on the board. The configureUART() method accomplishing this is seen in Appendix B.

Within the 'main' C module the configuration methods are run from the boardSupport.c file after the program is started. A function, getCharacter(), was then written to handle the UART input data.



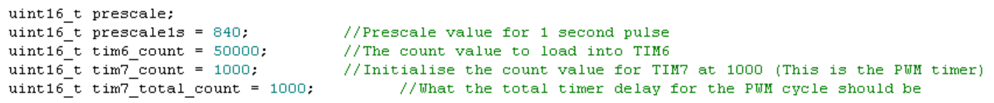
```
int8_t getCharacter()
{
  int8_t recieved = -1;
  int8_t valid= -1;
  // Check if a character has been recieved
  if(USART3->SR & USART_SR_RXNE)
  {
    //Get the character from the data register
    recieved = USART3->DR;

    //check if the character recieved was valid
    //Valid number check
    if (recieved >= '0' && recieved <= '9') valid = 1;
    //Control characters
    else if ((recieved == 0x0D) || (recieved == 0x0A)) valid = 1;
  }
  if (valid == -1) recieved = -1;
  return recieved;
}
```

Figure 3. getCharacter() method.

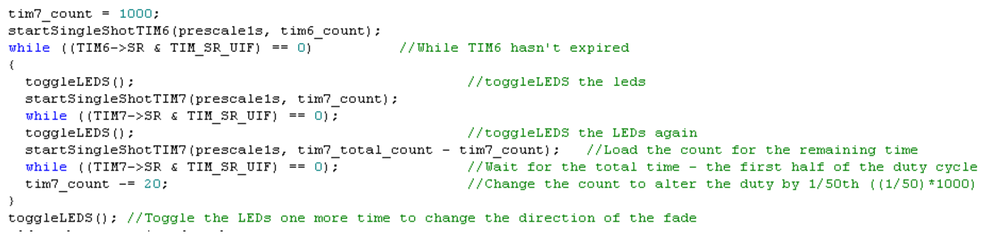
When called this method checks for a newly received character, and if valid, will return that character. Valid characters for this laboratory will be ascii values for any number [0, 9] or the two control characters 0x0D or 0x0A. Comparisons to single characters can be done in C as it converts the character provided to its respective ASCII value.

Next the timing routine controlling the LED fade was programmed. The timers were enabled for the AHB1 clock using the enableTIM6TIM7() method, and new methods were created to load the timers called startSingleShotTIM6() and startSingleShotTIM7(). These new methods can be seen in Appendix C. The methods set the Timer 6 and Timer 7 to a specified count and pre-scale value provided as uint16_t type arguments. The timers are also set to 'single shot mode', where through the single shot register the timer can be configured to only reach the count once and set the flag with out automatically starting again. The beginning of the method clears all relevant registers to allow for a new single shot every time the method is called.

Using the 'startSingleShot' methods each timer could be loaded and run independently, and a loop was made to run the program. Firstly, calculations were made to find the values needed for the count and pre-scale of the timers.



```
uint16_t prescale;
uint16_t prescale1s = 840;          //Prescale value for 1 second pulse
uint16_t tim6_count = 50000;        //The count value to load into TIM6
uint16_t tim7_count = 1000;         //Initialise the count value for TIM7 at 1000 (This is the PWM timer)
uint16_t tim7_total_count = 1000;         //What the total timer delay for the PWM cycle should be
```



```
tim7_count = 1000;
startSingleShotTIM6(prescale1s, tim6_count);
while ((TIM6->SR & TIM_SR_UIF) == 0)          //While TIM6 hasn't expired
{
  toggleLEDS();                                 //toggleLEDS the leds
  startSingleShotTIM7(prescale1s, tim7_count);
  while ((TIM7->SR & TIM_SR_UIF) == 0);
  toggleLEDS();                                 //toggleLEDS the LEDs again
  startSingleShotTIM7(prescale1s, tim7_total_count - tim7_count);   //Load the count for the remaining time
  while ((TIM7->SR & TIM_SR_UIF) == 0);         //Wait for the total time - the first half of the duty cycle
  tim7_count -= 20;                             //Change the count to alter the duty by 1/50th ((1/50)*1000)
}
toggleLEDS(); //Toggle the LEDs one more time to change the direction of the fade
```

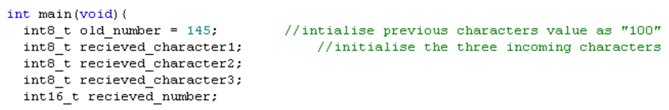
Figure 4. Fader timing loop.

As seen in Figure 4, the first while loop was implemented using the state of Timer 6 which is just anabled for a single shot. Calculations were made first to run the fade for one second, and it was intended that this first while loop would last for the duration of a whole second. To use easily divided numbers it was intended to run the clock at 50kHz, so the pre-scale was calculated as follows.

$$pre = \frac{42 * 10^6}{50 * 10^3} = 840$$

Therefore, with a pre-scale of 840 and a count of 50,000, Timer 6 would operate for one second. During this second the while loop is active, as and statement of the TIM6 state register and the 'timer finished' mask would be false. Within the while loop the LEDs are first toggled using a toggleLEDS() method which can be found in Appendix D. The toggleLEDS() method reads the current LED output data register (ODR) and performs an XOR operation with the mask, resulting in the inversion of the values. Therefore, resulting in a toggle on/off for the LEDS when the method is called. In Figure 4 we can see after the initial LED toggle Timer 7 is set and waited on by an empty while loop, therefore controlling the first half of the duty cycle. The LEDs are toggled again, and timer 7 loaded and waited on with a new count to control the latter end of the duty cycle. The same pre-scale value of 840 was chosen for Timer 7 for simplicity in calculations. A value named tim7_total_count was set to 1000 to signify the total time to be spent per-loop in the Timer 6 while loop and would be used for the PWM calculations each iteration. Each iteration of the loop sees the first Timer 7 wait of the loop reduced by 20 cycles, and the second Timer 7 wait increased by 20 cycles. This was calculated from the number of times the loop should complete within the Timer 6 loop which is 50,000/1,000 = 50 times. Therefore, the duty will gradually change from 100% to 0% in one second with a change in duty of twenty cycles per iteration. Finally, after the Timer 6 is complete, the LEDs are toggled again outside the Timer 6 while loop, such that they will start the while loop again in a different state. This will allow the LEDs to fade in on one whole cycle, and then fade out on the next, because the starting state for the leading duty cycle will have been toggled. The entire main() method can be seen in Appendix E.

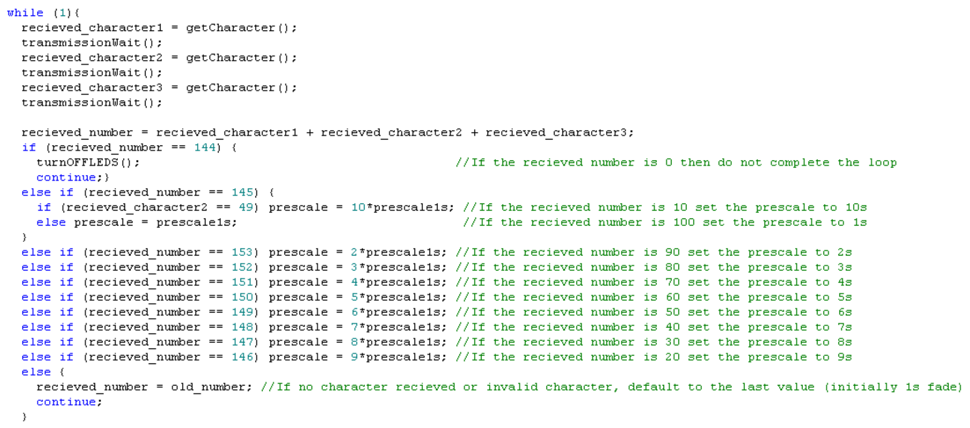
It was decided that the most effective way to adjust the fade time of the LEDs was to change the pre-scale value for both the timers. This way, the calculations for the duty cycle and the counts for the timers would not have to be touched, and time could effectively be scaled down or 'stretched' by adjusting the pre-scale. Given the calculations were already complete for a one-second fade, it is an easy multiplication of this value and the number of seconds the fade is wished to run for. Therefore, all that is left to accomplish is an input handler that will take the UART characters and apply the correct pre-scale multiplication at the start of each of the while(1) loops within the main method.



```
int main(void){
  int8_t old_number = 145;        //intialise previous characters value as "100"
  int8_t recieved_character1;         //initialise the three incoming characters
  int8_t recieved_character2;
  int8_t recieved_character3;
  int16_t recieved_number;
```

Figure 4. UART input variables declared.

Figure 4 shows the declaration of variables at the start of the main method ready to handle the input from the UART. One int8_t for each expected character, and one int16_t which will be used to sum the ASCII value of all the inputted numbers.

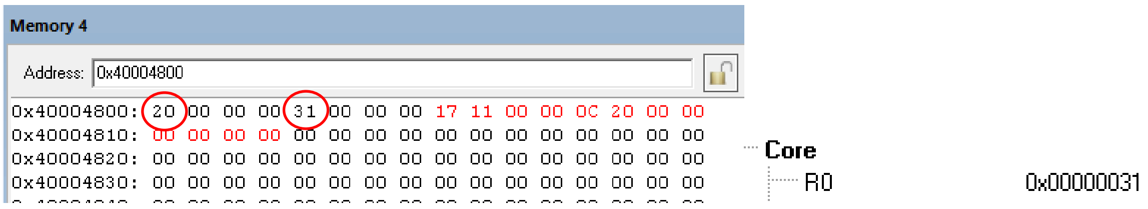


```
while (1){
  recieved_character1 = getCharacter();
  transmissionWait();
  recieved_character2 = getCharacter();
  transmissionWait();
  recieved_character3 = getCharacter();
  transmissionWait();

  recieved_number = recieved_character1 + recieved_character2 + recieved_character3;
  if (recieved_number == 144) {
    turnOFFLEDS();                                  //If the recieved number is 0 then do not complete the loop
    continue;}
  else if (recieved_number == 145) {
    if (recieved_character2 == 49) prescale = 10*prescale1s; //If the recieved number is 10 set the prescale to 10s
    else prescale = prescale1s;                              //If the recieved number is 100 set the prescale to 1s
  }
  else if (recieved_number == 153) prescale = 2*prescale1s; //If the recieved number is 90 set the prescale to 2s
  else if (recieved_number == 152) prescale = 3*prescale1s; //If the recieved number is 80 set the prescale to 3s
  else if (recieved_number == 151) prescale = 4*prescale1s; //If the recieved number is 70 set the prescale to 4s
  else if (recieved_number == 150) prescale = 5*prescale1s; //If the recieved number is 60 set the prescale to 5s
  else if (recieved_number == 149) prescale = 6*prescale1s; //If the recieved number is 50 set the prescale to 6s
  else if (recieved_number == 148) prescale = 7*prescale1s; //If the recieved number is 40 set the prescale to 7s
  else if (recieved_number == 147) prescale = 8*prescale1s; //If the recieved number is 30 set the prescale to 8s
  else if (recieved_number == 146) prescale = 9*prescale1s; //If the recieved number is 20 set the prescale to 9s
  else {
    recieved_number = old_number; //If no character recieved or invalid character, default to the last value (initially 1s fade)
    continue;
  }
```

Figure 5. UART Input Handling.

The main method sets the three received_character variables using the getCharacter() method, and waits each time using transmissionWait(). The transmission wait method is a simple method consisting of ten "__ASM("NOP");" instructions, allowing ten clock cycles for the next UART information to be transmitted. The received_number variable is then set to the sum of the three ASCII characters received. Given two of the received sums will be the same, for "010" and "100", the check for the input sum of 145 is checked twice for the value of the middle character to determine what was sent. For each if statement the pre-scale is set accordingly, and all unwanted sums will result in the 'else' branch where the pre-scale reverts to the previous value old_number (initialised as 153 for a UART input of '090'). The LEDs will remain off if the value received is '000' by using a method turnOFFLEDS() and continuing the loop. The method turnOFFLEDS() simply uses "&= ~" with the ODR mask on the ODR, setting the relevant bits off.

Given the Timer 6 and Timer 7 clocks are on the device's APB1 bus and this course is being taught remotely, simulation is limited. However, the UART was simulated using the Keil simulator and the while loop in the main function was successfully navigated to the correct if statement and pre-scale.

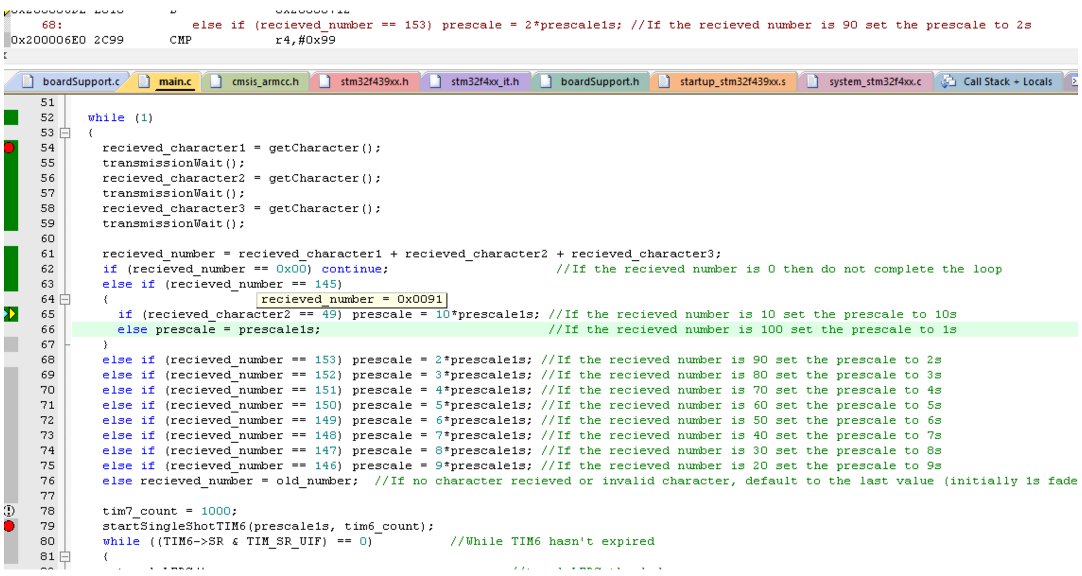


```
0x40004800: 20 00 00 00 31 00 00 00 17 11 00 00 0C 20 00 00
0x40004810: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0x40004820: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0x40004830: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

Figure 6. getCharacter() UART simulation and return.

Figure 6 shows on the left the USART register. It was loaded manually with the values circled, 0x02 to signal a received message and 0x31 as the data of that message. At the end of the getCharacter() method, R0 (the internal return register) was correctly loaded with the received hex value 0x31 as seen on the right of Figure 6.

This process was repeated three times with stepping through the code of the main() method, loading the UART data register with 0x31, 0x30, 0x30 respectively ('1', '0', '0' to form '100').



```
    68:              else if (recieved_number == 153) prescale = 2*prescale1s; //If the recieved number is 90 set the prescale to 2s
0x200006E0 2C99      CMP           r4,#0x99
```

```
  while (1)
  {
    recieved_character1 = getCharacter();
    transmissionWait();
    recieved_character2 = getCharacter();
    transmissionWait();
    recieved_character3 = getCharacter();
    transmissionWait();

    recieved_number = recieved_character1 + recieved_character2 + recieved_character3;
    if (recieved_number == 0x00) continue;                      //If the recieved number is 0 then do not complete the loop
    else if (recieved_number == 145)
    {               recieved_number = 0x0091
      if (recieved_character2 == 49) prescale = 10*prescale1s; //If the recieved number is 10 set the prescale to 10s
      else prescale = prescale1s;                              //If the recieved number is 100 set the prescale to 1s
    }
    else if (recieved_number == 153) prescale = 2*prescale1s; //If the recieved number is 90 set the prescale to 2s
    else if (recieved_number == 152) prescale = 3*prescale1s; //If the recieved number is 80 set the prescale to 3s
    else if (recieved_number == 151) prescale = 4*prescale1s; //If the recieved number is 70 set the prescale to 4s
    else if (recieved_number == 150) prescale = 5*prescale1s; //If the recieved number is 60 set the prescale to 5s
    else if (recieved_number == 149) prescale = 6*prescale1s; //If the recieved number is 50 set the prescale to 6s
    else if (recieved_number == 148) prescale = 7*prescale1s; //If the recieved number is 40 set the prescale to 7s
    else if (recieved_number == 147) prescale = 8*prescale1s; //If the recieved number is 30 set the prescale to 8s
    else if (recieved_number == 146) prescale = 9*prescale1s; //If the recieved number is 20 set the prescale to 9s
    else recieved_number = old_number;  //If no character recieved or invalid character, default to the last value (initially 1s fade

    tim7_count = 1000;
    startSingleShotTIM6(prescale1s, tim6_count);
    while ((TIM6->SR & TIM_SR_UIF) == 0)          //While TIM6 hasn't expired
    {
```

Figure 8. Simulated main() method.

The method program was then stepped through and checked for the right input handling. The correct if statement was entered as seen in Figure 8, and the recieved_number value equaled 0x91 = 145 as expected.

The Timer loops were unable to be simulated accordingly, however given the calculations and configuration as guided by the lecture material they should be able to be deployed and perform expectedly.

Conclusion

This investigation successfully implemented a UART receiving system as confirmed with simulation data. The timer routine was calculated and should perform as expected when deployed on the RMIT development board, however, no deployment or checks were able to be conducted due to the remote nature of this laboratory. The investigation required extensive use of C to apply the foundational knowledge gained through the course and previous labs, and successfully showed how C can be used as a powerful too for low-level programming. The lab also required the students to create a design for a PWM routine using internal timers within the STM32F439, customizing their configuration and operation with C methods being implemented to set the respective registers. This was an excellent learning exercise in the capabilities of embedded system implementation and was an excellent progression from the ARM assembly to now more functional C code. Using the initial flow chart, the program was created to the same design, and the laboratory can be regarded as a successful learning tool that required extensive demonstration of knowledge on the UART and Timer systems to be completed effectively.

Resources

[1] ST Microelectronics, "RM0090 – Reference Manual", ST Microelectronics, June 2018, Available Online.

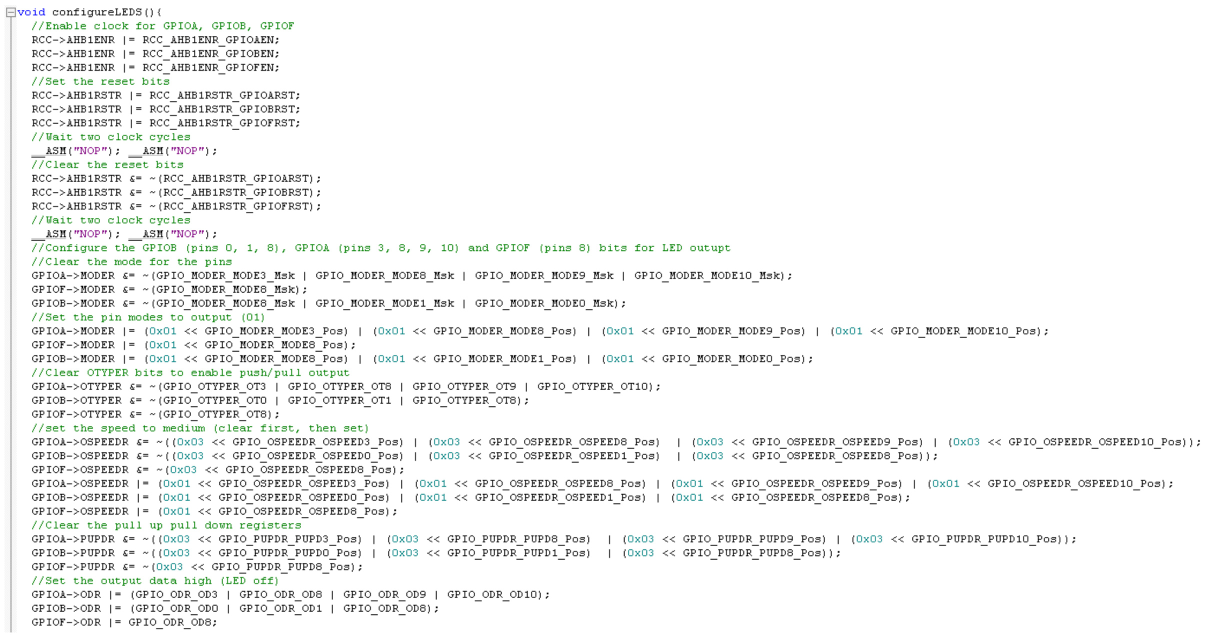
[2] ST Microelectronics, "PM0214 – Programming Manual", ST Microelectronics, February 2019, Available Online.

[3] G. Matthews, Class Lecture, "Introduction to Timer Peripherals." EEET2096, RMIT, Melbourne, April 2021.

[3] G. Matthews, Class Lecture, "Serial Communication - UART" EEET2096, RMIT, Melbourne, April 2021.

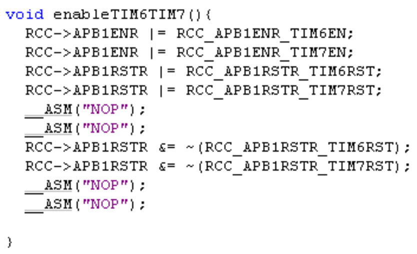
Appendix A

configureLEDS() code



```
void configureLEDS(){
  //Enable clock for GPIOA, GPIOB, GPIOF
  RCC->AHB1ENR |= RCC_AHB1ENR_GPIOAEN;
  RCC->AHB1ENR |= RCC_AHB1ENR_GPIOBEN;
  RCC->AHB1ENR |= RCC_AHB1ENR_GPIOFEN;
  //Set the reset bits
  RCC->AHB1RSTR |= RCC_AHB1RSTR_GPIOARST;
  RCC->AHB1RSTR |= RCC_AHB1RSTR_GPIOBRST;
  RCC->AHB1RSTR |= RCC_AHB1RSTR_GPIOFRST;
  //Wait two clock cycles
  __ASM("NOP"); __ASM("NOP");
  //Clear the reset bits
  RCC->AHB1RSTR &= ~(RCC_AHB1RSTR_GPIOARST);
  RCC->AHB1RSTR &= ~(RCC_AHB1RSTR_GPIOBRST);
  RCC->AHB1RSTR &= ~(RCC_AHB1RSTR_GPIOFRST);
  //Wait two clock cycles
  __ASM("NOP"); __ASM("NOP");
  //Configure the GPIOB (pins 0, 1, 8), GPIOA (pins 3, 8, 9, 10) and GPIOF (pins 8) bits for LED outupt
  //Clear the mode for the pins
  GPIOA->MODER &= ~(GPIO_MODER_MODE3_Msk | GPIO_MODER_MODE8_Msk | GPIO_MODER_MODE9_Msk | GPIO_MODER_MODE10_Msk);
  GPIOF->MODER &= ~(GPIO_MODER_MODE8_Msk);
  GPIOB->MODER &= ~(GPIO_MODER_MODE8_Msk | GPIO_MODER_MODE1_Msk | GPIO_MODER_MODE0_Msk);
  //Set the pin modes to output (01)
  GPIOA->MODER |= (0x01 << GPIO_MODER_MODE3_Pos) | (0x01 << GPIO_MODER_MODE8_Pos) | (0x01 << GPIO_MODER_MODE9_Pos) | (0x01 << GPIO_MODER_MODE10_Pos);
  GPIOF->MODER |= (0x01 << GPIO_MODER_MODE8_Pos);
  GPIOB->MODER |= (0x01 << GPIO_MODER_MODE8_Pos) | (0x01 << GPIO_MODER_MODE1_Pos) | (0x01 << GPIO_MODER_MODE0_Pos);
  //Clear OTYPER bits to enable push/pull output
  GPIOA->OTYPER &= ~(GPIO_OTYPER_OT3 | GPIO_OTYPER_OT8 | GPIO_OTYPER_OT9 | GPIO_OTYPER_OT10);
  GPIOB->OTYPER &= ~(GPIO_OTYPER_OT0 | GPIO_OTYPER_OT1 | GPIO_OTYPER_OT8);
  GPIOF->OTYPER &= ~(GPIO_OTYPER_OT8);
  //set the speed to medium (clear first, then set)
  GPIOA->OSPEEDR &= ~((0x03 << GPIO_OSPEEDR_OSPEED3_Pos) | (0x03 << GPIO_OSPEEDR_OSPEED8_Pos)  | (0x03 << GPIO_OSPEEDR_OSPEED9_Pos) | (0x03 << GPIO_OSPEEDR_OSPEED10_Pos));
  GPIOB->OSPEEDR &= ~((0x03 << GPIO_OSPEEDR_OSPEED0_Pos) | (0x03 << GPIO_OSPEEDR_OSPEED1_Pos)  | (0x03 << GPIO_OSPEEDR_OSPEED8_Pos));
  GPIOF->OSPEEDR &= ~(0x03 << GPIO_OSPEEDR_OSPEED8_Pos);
  GPIOA->OSPEEDR |= (0x01 << GPIO_OSPEEDR_OSPEED3_Pos) | (0x01 << GPIO_OSPEEDR_OSPEED8_Pos) | (0x01 << GPIO_OSPEEDR_OSPEED9_Pos) | (0x01 << GPIO_OSPEEDR_OSPEED10_Pos);
  GPIOB->OSPEEDR |= (0x01 << GPIO_OSPEEDR_OSPEED0_Pos) | (0x01 << GPIO_OSPEEDR_OSPEED1_Pos) | (0x01 << GPIO_OSPEEDR_OSPEED8_Pos);
  GPIOF->OSPEEDR |= (0x01 << GPIO_OSPEEDR_OSPEED8_Pos);
  //Clear the pull up pull down registers
  GPIOA->PUPDR &= ~((0x03 << GPIO_PUPDR_PUPD3_Pos) | (0x03 << GPIO_PUPDR_PUPD8_Pos)  | (0x03 << GPIO_PUPDR_PUPD9_Pos) | (0x03 << GPIO_PUPDR_PUPD10_Pos));
  GPIOB->PUPDR &= ~((0x03 << GPIO_PUPDR_PUPD0_Pos) | (0x03 << GPIO_PUPDR_PUPD1_Pos)  | (0x03 << GPIO_PUPDR_PUPD8_Pos));
  GPIOF->PUPDR &= ~(0x03 << GPIO_PUPDR_PUPD8_Pos);
  //Set the output data high (LED off)
  GPIOA->ODR |= (GPIO_ODR_OD3 | GPIO_ODR_OD8 | GPIO_ODR_OD9 | GPIO_ODR_OD10);
  GPIOB->ODR |= (GPIO_ODR_OD0 | GPIO_ODR_OD1 | GPIO_ODR_OD8);
  GPIOF->ODR |= GPIO_ODR_OD8;
```

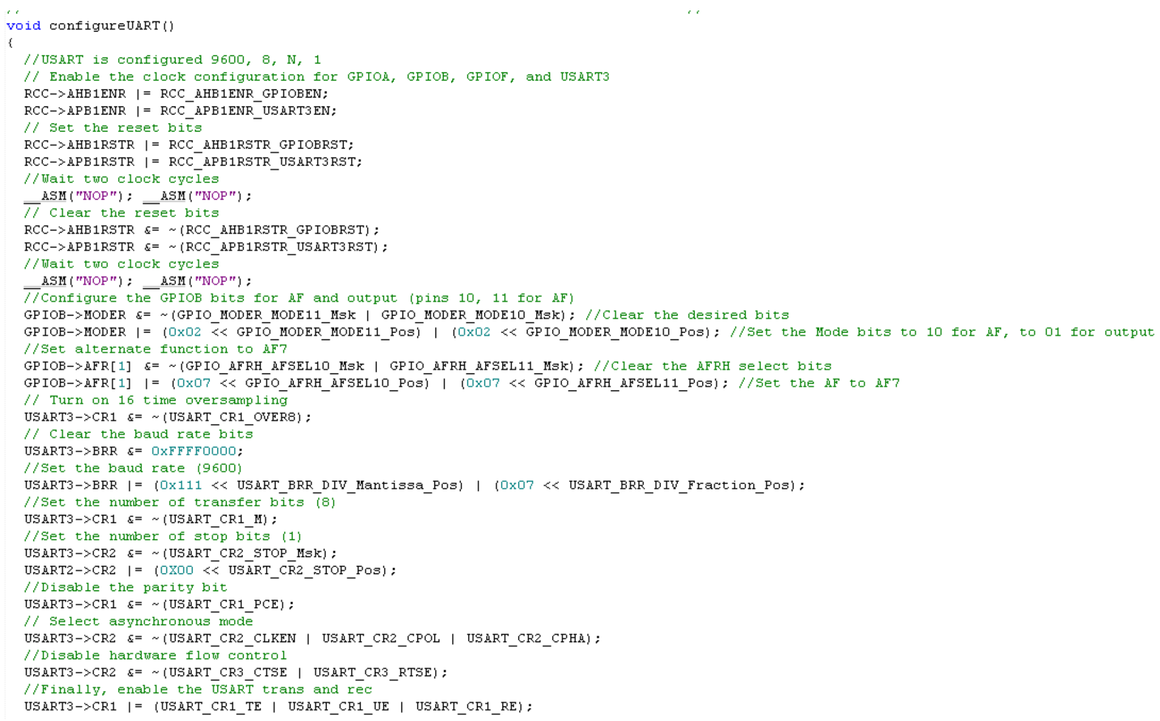
enableTIM6TIM7()



```
void enableTIM6TIM7(){
  RCC->APB1ENR |= RCC_APB1ENR_TIM6EN;
  RCC->APB1ENR |= RCC_APB1ENR_TIM7EN;
  RCC->APB1RSTR |= RCC_APB1RSTR_TIM6RST;
  RCC->APB1RSTR |= RCC_APB1RSTR_TIM7RST;
  __ASM("NOP");
  __ASM("NOP");
  RCC->APB1RSTR &= ~(RCC_APB1RSTR_TIM6RST);
  RCC->APB1RSTR &= ~(RCC_APB1RSTR_TIM7RST);
  __ASM("NOP");
  __ASM("NOP");

}
```

Appendix B

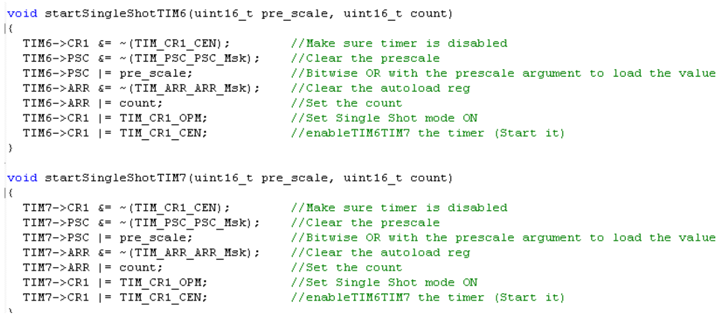
configureUART()



```
void configureUART()
{
  //USART is configured 9600, 8, N, 1
  // Enable the clock configuration for GPIOA, GPIOB, GPIOF, and USART3
  RCC->AHB1ENR |= RCC_AHB1ENR_GPIOBEN;
  RCC->APB1ENR |= RCC_APB1ENR_USART3EN;
  // Set the reset bits
  RCC->AHB1RSTR |= RCC_AHB1RSTR_GPIOBRST;
  RCC->APB1RSTR |= RCC_APB1RSTR_USART3RST;
  //Wait two clock cycles
  __ASM("NOP"); __ASM("NOP");
  // Clear the reset bits
  RCC->AHB1RSTR &= ~(RCC_AHB1RSTR_GPIOBRST);
  RCC->APB1RSTR &= ~(RCC_APB1RSTR_USART3RST);
  //Wait two clock cycles
  __ASM("NOP"); __ASM("NOP");
  //Configure the GPIOB bits for AF and output (pins 10, 11 for AF)
  GPIOB->MODER &= ~(GPIO_MODER_MODE11_Msk | GPIO_MODER_MODE10_Msk); //Clear the desired bits
  GPIOB->MODER |= (0x02 << GPIO_MODER_MODE11_Pos) | (0x02 << GPIO_MODER_MODE10_Pos); //Set the Mode bits to 10 for AF, to 01 for output
  //Set alternate function to AF7
  GPIOB->AFR[1]  &= ~(GPIO_AFRH_AFSEL10_Msk | GPIO_AFRH_AFSEL11_Msk); //Clear the AFRH select bits
  GPIOB->AFR[1]  |= (0x07 << GPIO_AFRH_AFSEL10_Pos) | (0x07 << GPIO_AFRH_AFSEL11_Pos); //Set the AF to AF7
  // Turn on 16 time oversampling
  USART3->CR1 &= ~(USART_CR1_OVER8);
  // Clear the baud rate bits
  USART3->BRR &= 0xFFFF0000;
  //Set the baud rate (9600)
  USART3->BRR |= (0x111 << USART_BRR_DIV_Mantissa_Pos) | (0x07 << USART_BRR_DIV_Fraction_Pos);
  //Set the number of transfer bits (8)
  USART3->CR1 &= ~(USART_CR1_M);
  //Set the number of stop bits (1)
  USART3->CR2 &= ~(USART_CR2_STOP_Msk);
  USART2->CR2 |= (0X00 << USART_CR2_STOP_Pos);
  //Disable the parity bit
  USART3->CR1 &= ~(USART_CR1_PCE);
  // Select asynchronous mode
  USART3->CR2 &= ~(USART_CR2_CLKEN | USART_CR2_CPOL | USART_CR2_CPHA);
  //Disable hardware flow control
  USART3->CR2 &= ~(USART_CR3_CTSE | USART_CR3_RTSE);
  //Finally, enable the USART trans and rec
  USART3->CR1 |= (USART_CR1_TE | USART_CR1_UE | USART_CR1_RE);
```

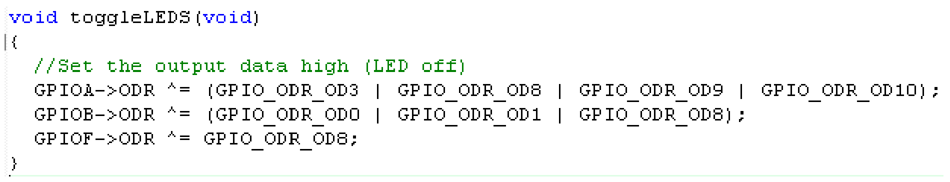
Appendix C

startSingleShotTIM6(pre_scale, count) and startSingleShotTIM7(pre_scale, count)



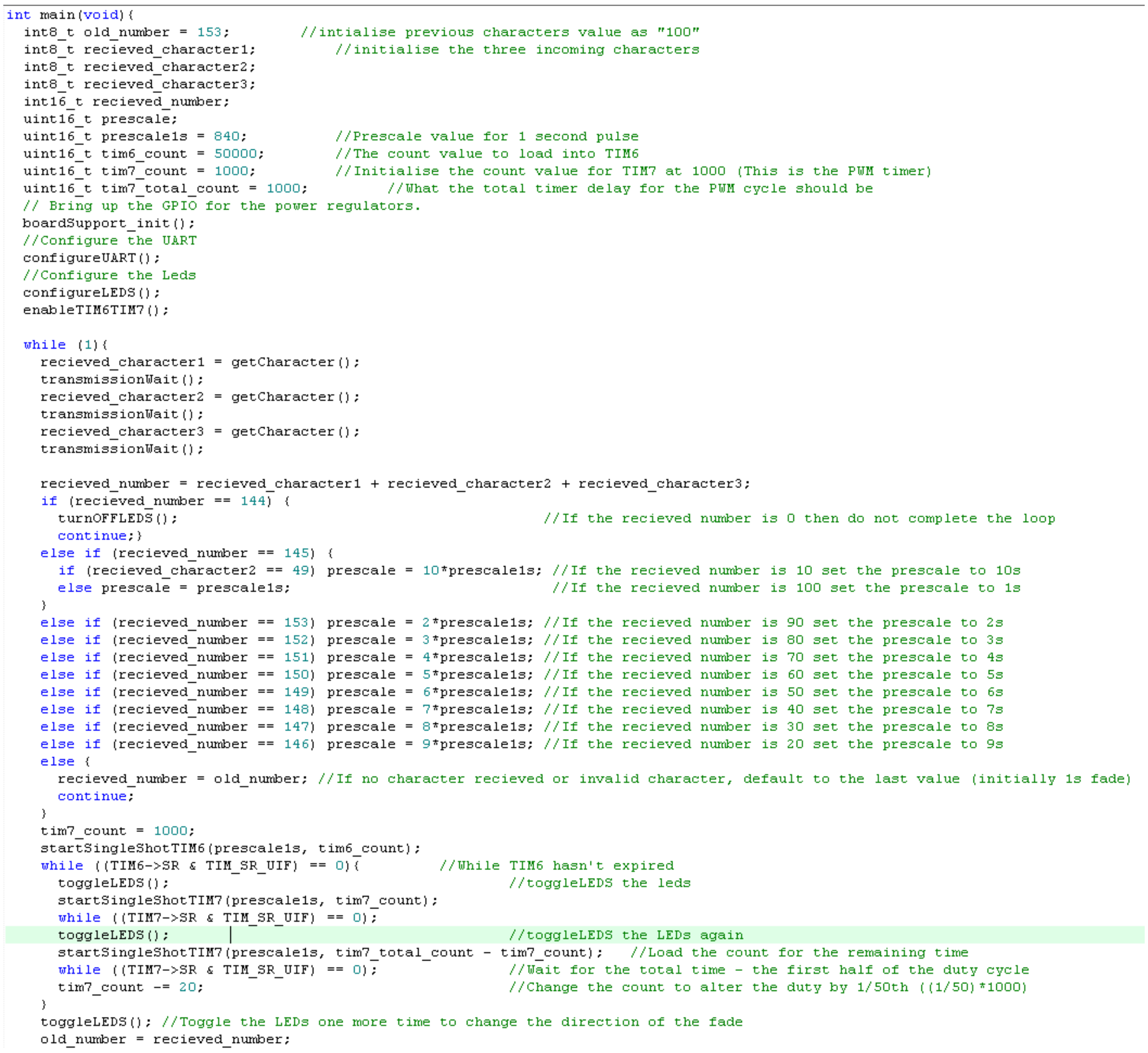
```
void startSingleShotTIM6(uint16_t pre_scale, uint16_t count)
{
  TIM6->CR1 &= ~(TIM_CR1_CEN);        //Make sure timer is disabled
  TIM6->PSC &= ~(TIM_PSC_PSC_Msk);    //Clear the prescale
  TIM6->PSC |= pre_scale;             //Bitwise OR with the prescale argument to load the value
  TIM6->ARR &= ~(TIM_ARR_ARR_Msk);    //Clear the autoload reg
  TIM6->ARR |= count;                 //Set the count
  TIM6->CR1 |= TIM_CR1_OPM;           //Set Single Shot mode ON
  TIM6->CR1 |= TIM_CR1_CEN;           //enableTIM6TIM7 the timer (Start it)
}

void startSingleShotTIM7(uint16_t pre_scale, uint16_t count)
{
  TIM7->CR1 &= ~(TIM_CR1_CEN);        //Make sure timer is disabled
  TIM7->PSC &= ~(TIM_PSC_PSC_Msk);    //Clear the prescale
  TIM7->PSC |= pre_scale;             //Bitwise OR with the prescale argument to load the value
  TIM7->ARR &= ~(TIM_ARR_ARR_Msk);    //Clear the autoload reg
  TIM7->ARR |= count;                 //Set the count
  TIM7->CR1 |= TIM_CR1_OPM;           //Set Single Shot mode ON
  TIM7->CR1 |= TIM_CR1_CEN;           //enableTIM6TIM7 the timer (Start it)
```

Appendix D



```
void toggleLEDS(void)
{
  //Set the output data high (LED off)
  GPIOA->ODR ^= (GPIO_ODR_OD3 | GPIO_ODR_OD8 | GPIO_ODR_OD9 | GPIO_ODR_OD10);
  GPIOB->ODR ^= (GPIO_ODR_OD0 | GPIO_ODR_OD1 | GPIO_ODR_OD8);
  GPIOF->ODR ^= GPIO_ODR_OD8;
}
```

Appendix E



```
int main(void){
  int8_t old_number = 153;        //intialise previous characters value as "100"
  int8_t recieved_character1;         //initialise the three incoming characters
  int8_t recieved_character2;
  int8_t recieved_character3;
  int16_t recieved_number;
  uint16_t prescale;
  uint16_t prescale1s = 840;          //Prescale value for 1 second pulse
  uint16_t tim6_count = 50000;        //The count value to load into TIM6
  uint16_t tim7_count = 1000;         //Initialise the count value for TIM7 at 1000 (This is the PWM timer)
  uint16_t tim7_total_count = 1000;         //What the total timer delay for the PWM cycle should be
  // Bring up the GPIO for the power regulators.
  boardSupport_init();
  //Configure the UART
  configureUART();
  //Configure the Leds
  configureLEDS();
  enableTIM6TIM7();

  while (1){
    recieved_character1 = getCharacter();
    transmissionWait();
    recieved_character2 = getCharacter();
    transmissionWait();
    recieved_character3 = getCharacter();
    transmissionWait();

    recieved_number = recieved_character1 + recieved_character2 + recieved_character3;
    if (recieved_number == 144) {
      turnOFFLEDS();                                      //If the recieved number is 0 then do not complete the loop
      continue;}
    else if (recieved_number == 145) {
      if (recieved_character2 == 49) prescale = 10*prescale1s; //If the recieved number is 10 set the prescale to 10s
      else prescale = prescale1s;                              //If the recieved number is 100 set the prescale to 1s
    }
    else if (recieved_number == 153) prescale = 2*prescale1s; //If the recieved number is 90 set the prescale to 2s
    else if (recieved_number == 152) prescale = 3*prescale1s; //If the recieved number is 80 set the prescale to 3s
    else if (recieved_number == 151) prescale = 4*prescale1s; //If the recieved number is 70 set the prescale to 4s
    else if (recieved_number == 150) prescale = 5*prescale1s; //If the recieved number is 60 set the prescale to 5s
    else if (recieved_number == 149) prescale = 6*prescale1s; //If the recieved number is 50 set the prescale to 6s
    else if (recieved_number == 148) prescale = 7*prescale1s; //If the recieved number is 40 set the prescale to 7s
    else if (recieved_number == 147) prescale = 8*prescale1s; //If the recieved number is 30 set the prescale to 8s
    else if (recieved_number == 146) prescale = 9*prescale1s; //If the recieved number is 20 set the prescale to 9s
    else {
      recieved_number = old_number; //If no character recieved or invalid character, default to the last value (initially 1s fade)
      continue;
    }
    tim7_count = 1000;
    startSingleShotTIM6(prescale1s, tim6_count);
    while ((TIM6->SR & TIM_SR_UIF) == 0){          //While TIM6 hasn't expired
      toggleLEDS();                                //toggleLEDS the leds
      startSingleShotTIM7(prescale1s, tim7_count);
      while ((TIM7->SR & TIM_SR_UIF) == 0);
      toggleLEDS();                                //toggleLEDS the LEDs again
      startSingleShotTIM7(prescale1s, tim7_total_count - tim7_count);   //Load the count for the remaining time
      while ((TIM7->SR & TIM_SR_UIF) == 0);        //Wait for the total time - the first half of the duty cycle
      tim7_count -= 20;                            //Change the count to alter the duty by 1/50th ((1/50)*1000)
    }
    toggleLEDS(); //Toggle the LEDs one more time to change the direction of the fade
    old_number = recieved_number;
```