IR Remote Control Text Messaging

Laboratory 3 - EEC 172

Kendall Lui and Mary Florek

2/15/18

INTRODUCTION

A common technology used to communicate between two devices is an infrared (IR) receiver module. We used this receiver in this lab to send text messages from one OLED screen to another. Because the receiver works on infrared light, a direct line-of-sight is needed for it to work properly. As the infrared signal is detected on the receiver, we were able to use interrupts to help us decode the signal. Interrupts are signals emitted by the software or hardware in order to call attention to a certain event. The data from the interrupt is sent to the Interrupt Handler, which then takes action towards resolving the interrupt. Interrupts are a crucial subject for embedded system engineers as they are used to help systems successfully communicate based on external conditions.

GOALS

This lab is designed to further the understanding of embedded systems by the creation of an IR Decoder and Text Messaging app. The first step is understanding how IR remotes work and the protocols that are used to transmit data. The implementation of these projects give exposure to interrupts, timers, and UART communication configuration. In addition, this project required understanding how to interface interrupts with polling to create a reliable and responsive system. From a design standpoint, these projects required a detailed design phase to make each subsystem interact with each other seamlessly.

METHODS

The first step to implement the IR Remote Control Texting application was analyzing the IR signals of the remote code 1156. A 100-ohm resistor and 100 micro-farad capacitor were connected to the receiver to help reduce noise from the power source. The Saleae Logic analyzer was used with the Vishay IR receiver to display the signal produced by IR remote. It was determined that the 1156 signal was of the NEC IR transmission protocol as defined in the supporting documentation provided for the lab.

As can be seen in the screenshots attached at the end of this report, there is a start signal with a low for 9ms followed by a high with 4.5ms. The NEC code makes use of pulse distance modulation with a pulse width of 1.125ms representing a 1 and 2.25ms representing a 0 measured from rising edge to rising edge. The first 8 bits are the address code used to identify the source of the data. The next 8 bits are the inverted address which can be used to detect corruption. After the address the next 16 bits are the data and the inverted data.

The address was determined to be 0x02. After pressing several of the buttons and manually reading the data we found that keys 0-9 were directly mapped to data output of their corresponding value with the least significant bit transmitted first. Additionally, the data for the enter key is 0x17 and the data for the mute key is 0x10.

After understanding the encoding for the transmission, the next step was to implement a decoder for the CC3200 launchpad using interrupts. The interrupt was set to interrupt on the rising edge input from the Vishay receiver. On the interrupt if a timer was set the next rising edge interrupt would then store the elapsed time into a circular buffer.

The buffer was then processed by two functions. The first function, parseIRData(), converts the pulses into binary. It checks for the leading 4.5 ms and then converts the next 32 pulse times into bits using a simple shift operator. The data is then validated using the function getIRData() by checking the address and appropriate inverted data. The data was then printed to the UART terminal with special cases printing for the “ENTER” and “MUTE” keys.

With the IR decoder setup the next step was writing a texting program. The IR remote code was further developed to detect multiple key presses to implement keypad typing. The implementation simply tracks the number of times the button is pressed in a row. A 2D array is used to convert this count to a desired character. A timer interrupt is started each time the button is pressed to detect whether the current character should be stored. If the timer expires the code is interrupted and the character is stored and the cursor advanced.

After developing the typing tool, communication using UART1 was implemented to send messages to another board. Using the UART API provided by CC3200 the port was set up with a baud rate of 115200. Additionally, interrupt is registered to detect when a message is received. When this interrupt occurs it reads each character from the UART1 port.

A texting library was created called TextManager that managed the text message display, text message uart communication, and text message composer. When a character is first typed the IR library calls on placeCharacter() and when the character needs to be stored it calls advanceCursor(). A sendMessage() is called when enter is pressed to transmit over the UART1 buffer.

Finally, the OLED display was implemented within the TextManager library. A blinking cursor was created using a timer interrupt which simply changed the color of a null character from white to black. When a button is pressed the display shows the current character highlighted to aid users to see what they are typing. Clearing the display simply pastes null characters over the previously written text.

DISCUSSION

The first part of this lab, decoding the IR data was challenging since at first it was unclear what was being recorded. After reading the documents and looking closer at the signal it became clear that the signal was NEC. Once this was clear the specifications were very straightforward.

Implementing the IR receiver required setting up timers and interrupts. This proved to be challenging because there are a lot of different parameters that can be set on a timer from period to one shot timers. The timers give the number of clock ticks which can be used with the clock frequency to determine the actual time in seconds. Reading the documentation helped with understanding how the timers worked in detail with information that can be used in future projects. The IR implementation also accounts for a 20% error in the timing since the real world will not get the exact timing as specified by the protocol.

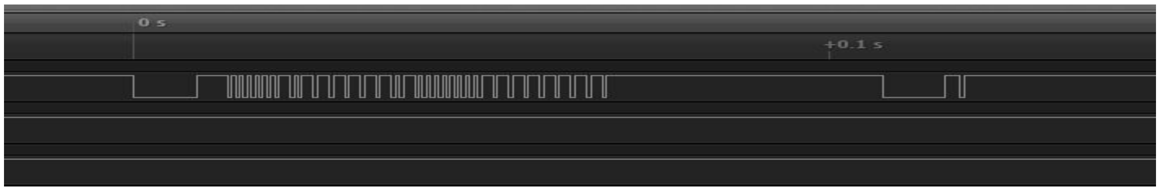
The last issue was implementing the UART1 configuration. This took a lot of time since documentation for UART1 is not widely available. Most resources demonstrated the use of UART0 and the different configuration parameters were poorly documented. Additionally, there was a problem with the complete UART data not being sent over. This is due to the way the FIFO buffer and interrupts were set up. In order to fix this problem the FIFO buffer was turned off. This solved the problem because the interrupt was only being received at a certain level the FIFO became full not at every time a message was received.

After implementing most of the features, the code was split up into IRDecoder and TextManager library files. This allows for more organized code and keeps things relatively separate in their functions. Cleaning up the code was performed to minimize the amount of time an interrupt service routine needed to handle the code. SPI is slow so this would cause the system to react slowly whenever it was being written to. Fixing this required using interrupts to set flags and store data. These were then polled in order to make a more responsive system.

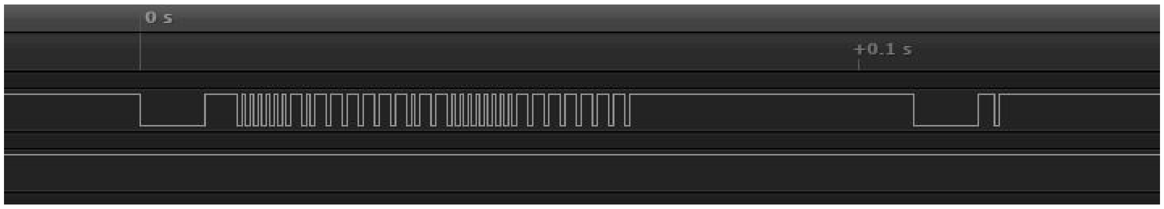
CONCLUSION

We were ultimately able to decode the IR data using the NEC signal specifications. Implementing a text messaging system was non-trivial but ultimately successful as well. We accomplished our initial goal of learning the protocols used in IR transmission, and the concept of interrupts.

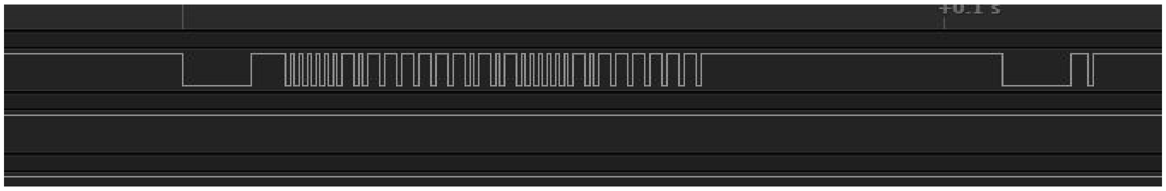
Binary Representation of 0:



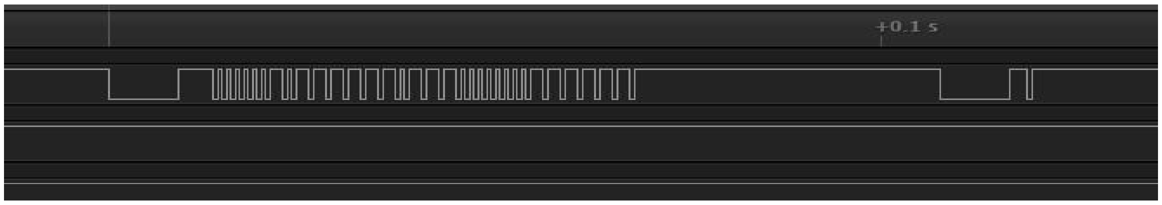
Binary Representation of 1:



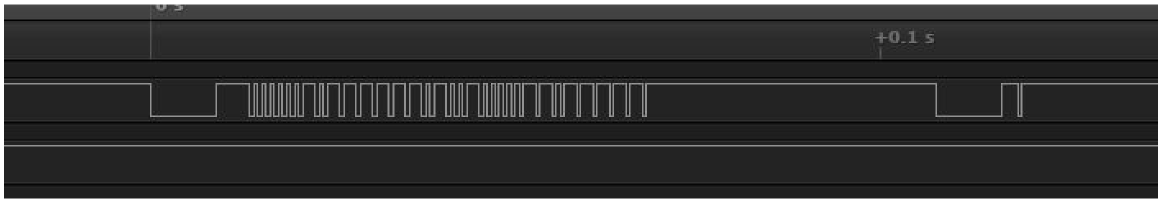
Binary Representation of 2:



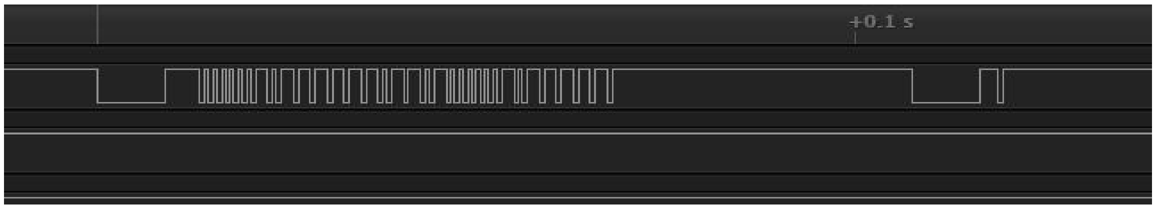
Binary Representation of 3:



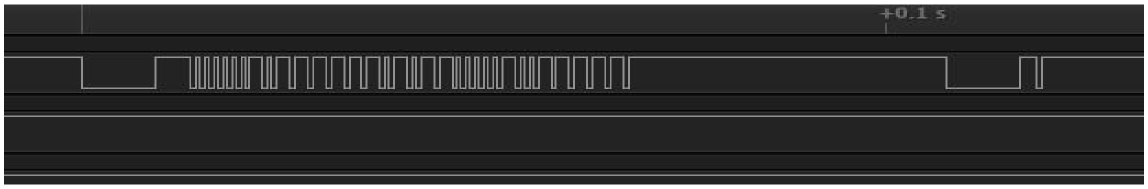
Binary Representation of 4:



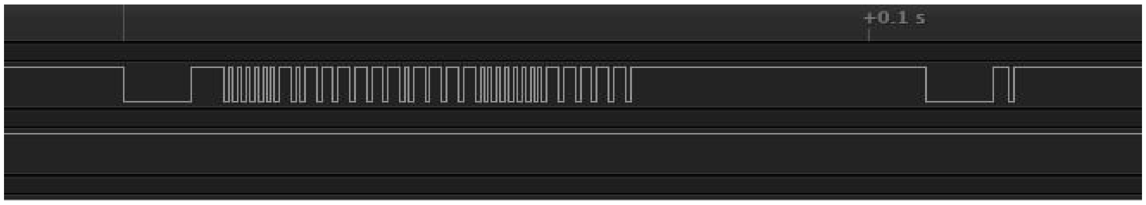
Binary Representation of 5:



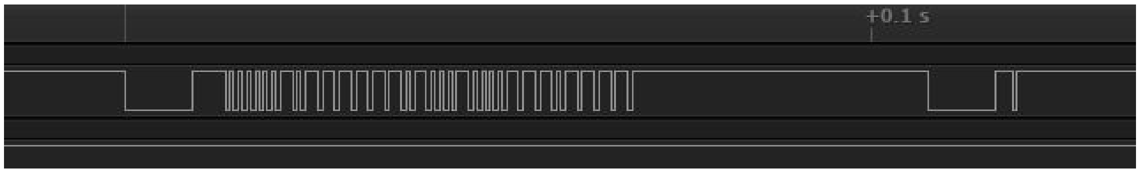
Binary Representation of 6:



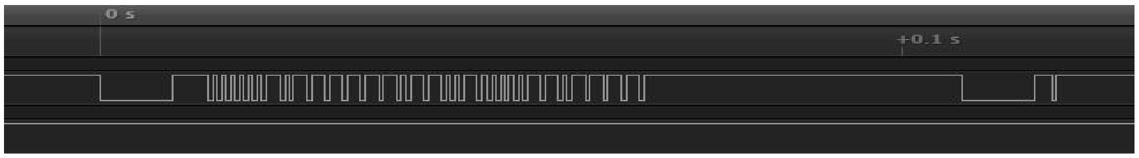
Binary Representation of 7:



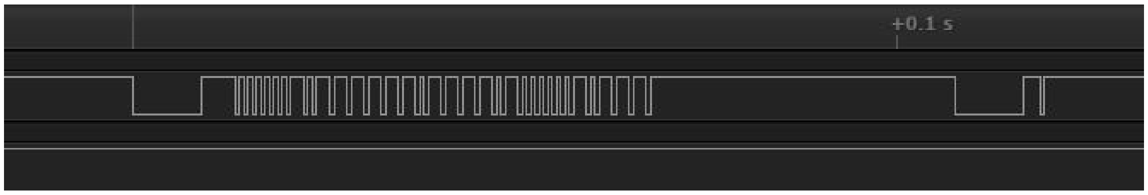
Binary Representation of 8:



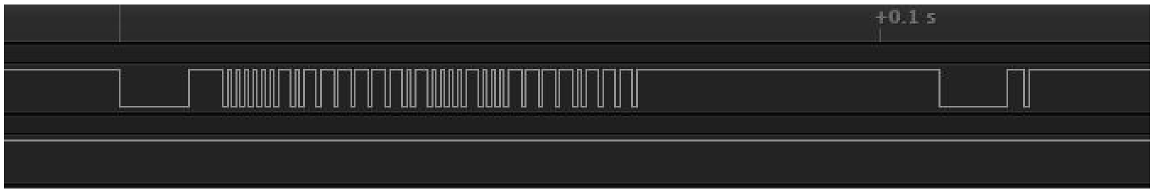
Binary Representation of 9:



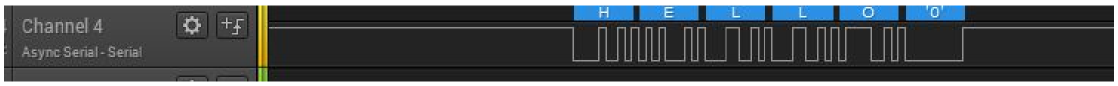
Binary Representation of “Enter”:



Binary Representation of “Mute”:



UART representation of “HELLO” with the Async Serial protocol analyzer



/*

```
 * IRDecoder.c
 *
 *  Created on: Feb 5, 2018
 *          Author: Kendall
 */
// Driverlib includes
#include "hw_types.h"
#include "hw_ints.h"
#include "hw_memmap.h"
#include "hw_common_reg.h"
#include "interrupt.h"
#include "hw_apps_rcm.h"
#include "prcm.h"
#include "rom.h"
#include "rom_map.h"
#include "prcm.h"
#include "gpio.h"
#include "utils.h"
#include "spi.h"


// Common interface includes
#include "timer.h"
#include "timer_if.h"

#include "TextManager.h"
#include "IRDecoder.h"

#define BUFFER_SIZE 100
//Pulse Count Upper and Lower limits to account for timing inaccuracies
#define LOWER_START 288000 // 20% margin
#define UPPER_START 432000 // 20% margin
#define LOWER_ZERO  72000  // 20% margin
#define UPPER_ZERO  108000 // 20% margin
#define LOWER_ONE   144000 // 20% margin
#define UPPER_ONE   216000 // 20% margin
#define ADDRESS 0xbf40

// Private Variables
volatile int bufferCount = 0;
volatile long int buffer[BUFFER_SIZE];
int currentBufferLocation = 0;

// Parsing Buffer Data into int
unsigned int IRData = 0; //actual IRdata
char startSignal = 0; // Signals that the bit data is starting
char bitCount = 0; // Counts to 32
char prevData = '\0';
char alpha[10][4] = {
                        {' ',' ',' ',' '},
                        {'.','!','?',','},
                        {'A', 'B', 'C','C'},
                        {'D', 'E', 'F','F'},
                        {'G', 'H', 'I','I'},
                        {'J', 'K', 'L','L'},
                        {'M', 'N', 'O','O'},
                        {'P', 'Q', 'R', 'S'},
                        {'T', 'U', 'V', 'V'},
                        {'W', 'X', 'Y', 'Z'},
            };

// State Flags
int consecutivePressFlag = 0;

static void RisingIntHandler(void);
void DecodeIR(void);
void ConsecutivePressTimerINT(void);


void IRSetupINT()
{
            //ISR
            MAP_GPIOIntRegister(GPIOA3_BASE, RisingIntHandler);
            MAP_GPIOIntTypeSet(GPIOA3_BASE, 0x80, GPIO_RISING_EDGE);
            MAP_GPIOIntClear(GPIOA3_BASE, MAP_GPIOIntStatus (GPIOA3_BASE, false));
            MAP_GPIOIntEnable(GPIOA3_BASE, 0x80);

            //
            // Configure the timer in Periodic Up
            // For use with IR Data Reader
            //
            TimerConfigure(TIMERA2_BASE, TIMER_CFG_PERIODIC_UP);
            TimerEnable(TIMERA2_BASE,TIMER_A);
            TimerValueSet(TIMERA2_BASE,  TIMER_A,0);

            //
            // Setup timer for consecutive
            Timer_IF_Init(PRCM_TIMERA0, TIMERA0_BASE, TIMER_CFG_ONE_SHOT, TIMER_A, 0);
            Timer_IF_IntSetup(TIMERA0_BASE, TIMER_A, ConsecutivePressTimerINT);

}


static void RisingIntHandler(void) { // IR handler
            unsigned long ulStatus;

            ulStatus = MAP_GPIOIntStatus (GPIOA3_BASE, true);
            MAP_GPIOIntClear(GPIOA3_BASE, ulStatus);                // clear interrupts on GPIOA3
```

```
	buffer[bufferCount] = TimerValueGet( TIMERA2_BASE,  TIMER_A  );
	//Circular Buffer
	if(bufferCount < BUFFER_SIZE-1) {
	bufferCount++;
	} else {
	bufferCount = 0;
	}

	TimerValueSet(TIMERA2_BASE,  TIMER_A,0);
}
void DecodeIR(void)
{
	if(dataReady != 1) {
	// Detects whether data is within the appropriate delay times
	if(buffer[currentBufferLocation] < UPPER_START && buffer[currentBufferLocation] > LOWER_START )
	{		// Start = 4.5ms
	startSignal = 1;
	bitCount = 0;
	IRData = 0;
	} else if(buffer[currentBufferLocation] < UPPER_ZERO && buffer[currentBufferLocation] > LOWER_ZERO)
	{		// Zero = 1.125ms
	if(startSignal == 1)
	{
	bitCount++;
	if(bitCount == 32) {
		dataReady = 1;
	}
	} else
	{ // Bad Data or Interference
	startSignal = 0;
	bitCount = 0;
	}
	} else if(buffer[currentBufferLocation] < UPPER_ONE && buffer[currentBufferLocation] > LOWER_ONE)
	{		// One = 2.25ms
	if(startSignal == 1) {
	IRData += (0x1 << bitCount);
	bitCount++;
	if(bitCount == 32) {
		dataReady = 1;
	}
	} else { // Bad Data or Interference
	startSignal = 0;
	bitCount = 0;
	}
	} else {  // If we get here the signal timed out.
	startSignal = 0;
	bitCount = 0;
	}
	// Increment currentBufferLocation and circular buffer
	if(currentBufferLocation < BUFFER_SIZE-1)
```

```
        {
        currentBufferLocation++;
        } else {
        currentBufferLocation = 0;
        }
        }
}
void ProcessIR(void)
{
        if(bufferCount != currentBufferLocation) {
        DecodeIR();

        }
        if(dataReady == 1) {
        GetIRData();
        }

}
void GetIRData(void) {
        if(ADDRESS == (IRData<<16)>>16) {
        char data = (IRData<<8)>>24;
        char inverted = ((~IRData)>>24);
        if(data == inverted) { //Checks for corrupt data
        if(data < 10) { //unsigned
        Timer_IF_Stop(TIMERA1_BASE, TIMER_A);
        if(data != prevData && consecutivePressFlag > 0)
        {
                Timer_IF_Stop(TIMERA0_BASE, TIMER_A);
                consecutivePressFlag = 0;
                advanceCursor();
        }
        prevData = data;
        placeCharacter( alpha[data][consecutivePressFlag]);
        if(consecutivePressFlag < 3) {
                consecutivePressFlag++;
                Timer_IF_Start(TIMERA0_BASE, TIMER_A, 700);
        } else {
                Timer_IF_Stop(TIMERA0_BASE, TIMER_A);
                consecutivePressFlag = 0;
                advanceCursor();
        }
        } else if(data == 0x17) {
        prevData = 0;
        advanceCursor();
        sendMessage();
        //Report("%s\n\r",textString);
        } else if(data == 0x10) {
        deleteChar();
        }
        }
```

```
		} // Checks Address
		dataReady = 0;
}
void ConsecutivePressTimerINT(void)
{
		//
		// Clear the timer interrupt.
		//
		Timer_IF_InterruptClear(TIMERA0_BASE);
		consecutivePressFlag = 0;
		advanceCursor();
}

/*
 * TextManager.c
 *
 *  Created on: Feb 5, 2018
 *		Author: Kendall
 */
#include "hw_types.h"
#include "hw_memmap.h"
#include "prcm.h"
#include "pin.h"
#include "uart.h"
#include "rom.h"
#include "rom_map.h"
#include "uart.h"
#include "timer.h"
#include "timer_if.h"
#include "Adafruit_GFX.h"
#include "Adafruit_SSD1351.h"

#include "TextManager.h"

//Private Functions
void BlinkCursor(void);

// States
static int cursorBlink = 0;

// Internal Storage
char stringMessage[160]= "";
int messageLength = 0;

//inbox
char inboxMessageLength = 0;

void SetupCommunication()
{
		PRCMPeripheralClkEnable(PRCM_UARTA1, PRCM_RUN_MODE_CLK);
	PRCMPeripheralClkEnable(PRCM_UARTA0, PRCM_RUN_MODE_CLK);

	PinTypeUART(PIN_58, PIN_MODE_6); //UART1_TX
	PinTypeUART(PIN_59, PIN_MODE_6); //UART1_RX

	PinTypeUART(PIN_55, PIN_MODE_3); //UART0_TX
	PinTypeUART(PIN_57, PIN_MODE_3); //UART0_RX

	//UART Setup
	UARTConfigSetExpClk(UARTA1_BASE, 80000000, 115200, (UART_CONFIG_WLEN_8 | UART_CONFIG_STOP_ONE |
UART_CONFIG_PAR_NONE));
	UARTConfigSetExpClk(UARTA0_BASE, 80000000, 115200, (UART_CONFIG_WLEN_8 | UART_CONFIG_STOP_ONE |
UART_CONFIG_PAR_NONE));

	UARTEnable(UARTA1_BASE);
	UARTEnable(UARTA0_BASE);
	UARTDMADisable  ( UARTA1_BASE, (UART_DMA_RX | UART_DMA_TX ));
	UARTFIFODisable  ( UARTA1_BASE ) ;
}

void SetupDisplay()
{
	Adafruit_Init(); // Start board
	fillScreen(Color565(0,0,0));

	//
	// Setup timer for blinking Cursor
	Timer_IF_Init(PRCM_TIMERA1, TIMERA1_BASE, TIMER_CFG_PERIODIC, TIMER_A, 0);
	Timer_IF_IntSetup(TIMERA1_BASE, TIMER_A, BlinkCursor);
	Timer_IF_Start(TIMERA1_BASE, TIMER_A, 500);

	UARTIntEnable( UARTA1_BASE,  UART_INT_RX) ;
	UARTIntRegister(UARTA1_BASE,checkInbox);
}

void BlinkCursor(void) {
	Timer_IF_InterruptClear(TIMERA1_BASE);
	if(cursorBlink == 0) {
	cursorBlink = 1;
	} else {
	cursorBlink = 0;
	}
}

int prevCursor = 0;
void displayCursor()
{
	if(cursorBlink != prevCursor)
	{
	if(cursorBlink == 0) {
		drawChar(6*(messageLength%21), 7* (messageLength/21), 0, Color565(255,255,255), Color565(0,0,0), 1);
		prevCursor = cursorBlink;
		} else {
		drawChar(6*(messageLength%21), 7* (messageLength/21), 0, Color565(0,0,0), Color565(255,255,255), 1);
		prevCursor = cursorBlink;
		}
		}

}

void placeCharacter(char c)
{
  Timer_IF_Stop(TIMERA1_BASE, TIMER_A);
  stringMessage[messageLength] = c;
  stringMessage[messageLength+1] = '\0';
  drawChar(6*(messageLength%21), 7* (messageLength/21), c, Color565(0,0,0), Color565(255,255,255), 1);
}

void advanceCursor()
{
		drawChar(6*(messageLength%21), 7* (messageLength/21), stringMessage[messageLength] , Color565(255,255,255), Color565(0,0,0), 1);
		messageLength++;
		Timer_IF_Start(TIMERA1_BASE, TIMER_A, 500);
}
char sendBuffer[160];
char sendBufferLength = 0;
char bufferState = 0;

void sendMessage()
{
		if(messageLength > 0) {
		int i;
		for(i = 0; i< messageLength; i++)
		{
		drawChar(6*(i%21), (7* (i/21)), 0, Color565(255,255,255), Color565(0,0,0), 1);
		}

		for(i =0; i < messageLength;i++)
		{
		while(UARTBusy( UARTA1_BASE ));
		UARTCharPut(UARTA1_BASE,stringMessage[i]);
		}
		UARTCharPut(UARTA1_BASE,'\0');
		messageLength = 0;
		stringMessage[0] = 0;
		}
}

char inboxMessage[160];
char messageReady = 0;
char prevMessageLength = 0;
void checkInbox(){
        UARTIntClear (UARTA1_BASE,UART_INT_RX);
        while(UARTCharsAvail( UARTA1_BASE ))
        {
        char c = UARTCharGet(UARTA1_BASE);
        if(c == '\0') {
        messageReady = 1;
        } else {
        inboxMessage[inboxMessageLength] = c;
        inboxMessageLength++;
        }
        }
}
void displayMessage()
{
        if(messageReady)
        {
        int i =0;
        for(i=0;i<inboxMessageLength; i++)
        {
        drawChar(6*(i%21), (8* (i/21))+58,inboxMessage[i] , Color565(0,255,255), Color565(0,0,0), 1);
        }
        for(i = inboxMessageLength;i<=prevMessageLength; i++)
        {
        drawChar(6*(i%21), (8*(i/21))+58, 0 , Color565(255,255,255), Color565(0,0,0), 1);
        }
        messageReady = 0;
        prevMessageLength = inboxMessageLength;
        inboxMessageLength = 0;
        }

}
void deleteChar()
{
        drawChar(6*(messageLength%21), 7* (messageLength/21), 0, Color565(255,255,255), Color565(0,0,0), 1);
        messageLength--;
        Timer_IF_Start(TIMERA1_BASE, TIMER_A, 500);
}

// Standard includes
#include <stdio.h>

// Driverlib includes
#include "hw_types.h"
#include "hw_ints.h"
#include "hw_memmap.h"
#include "hw_common_reg.h"
#include "interrupt.h"
#include "hw_apps_rcm.h"
```

```
#include "prcm.h"
#include "rom.h"
#include "rom_map.h"
#include "prcm.h"
#include "gpio.h"
#include "utils.h"
#include "spi.h"

// Common interface includes
#include "uart_if.h"
#include "timer.h"
#include "timer_if.h"
#include "pinmux.h"
#include "Adafruit_GFX.h"
#include "Adafruit_SSD1351.h"
#include "TextManager.h"
#include "IRDecoder.h"

#define BUFFER_SIZE 100

//Pulse Count Upper and Lower limits to account for timing inaccuracies
#define LOWER_START 288000 // 20% margin
#define UPPER_START 432000 // 20% margin
#define LOWER_ZERO  72000
#define UPPER_ZERO  108000
#define LOWER_ONE   144000
#define UPPER_ONE   216000
#define ADDRESS 0xbf40

//*****************************************************************************
//              GLOBAL VARIABLES -- Start
//*****************************************************************************
extern void (* const g_pfnVectors[])(void);

static void BoardInit(void);

//*****************************************************************************
//! Board Initialization & Configuration
//!
//! \param  None
//!
//! \return None
//*****************************************************************************
static void
BoardInit(void) {
        MAP_IntVTableBaseSet((unsigned long)&g_pfnVectors[0]);

        // Enable Processor
        //
        MAP_IntMasterEnable();
```

```
		MAP_IntEnable(FAULT_SYSTICK);

		PRCMCC3200MCUInit();
}
//*****************************************************************************
//! Main function
//!
//! \param none
//!
//! \return None.
//*****************************************************************************
int main() {

		BoardInit();

		PinMuxConfig();
		SetupCommunication();

		InitTerm();

		ClearTerm();

		Message("\t\t****************************************************\n\r");
		Message("\t\t\tIR Remote Decoder\n\r");
		Message("\t\t\tConfigured for remote control code 1156\n\r");
		Message("\t\t\tPress any button 0 - 9, ENTER, and MUTE\n\r");
		Message("\t\t ****************************************************\n\r");
		Message("\n\n\n\r");
		SetupDisplay();
		IRSetupINT();


		while (1) {
		ProcessIR();
		displayMessage();
		displayCursor();
		}


}
```