2303ENG Microprocessor Techniques

Infrared Messaging

A minimalistic infrared compatible messaging protocol

Gavin Wong s2896164 – Griffith University

23/10/2014

Table of Contents

Introduction

The aim of the project is to construct an infrared compatible messaging protocol, suitable for transmitting arbitrary data in a bi-directional manner. For the purpose of this project, the protocol will be consumed as a chat service, delivering textual messages between two parties. However it is not limited to just textual messages, it can be used for other purposes such as file transfer, remote controlling etc.

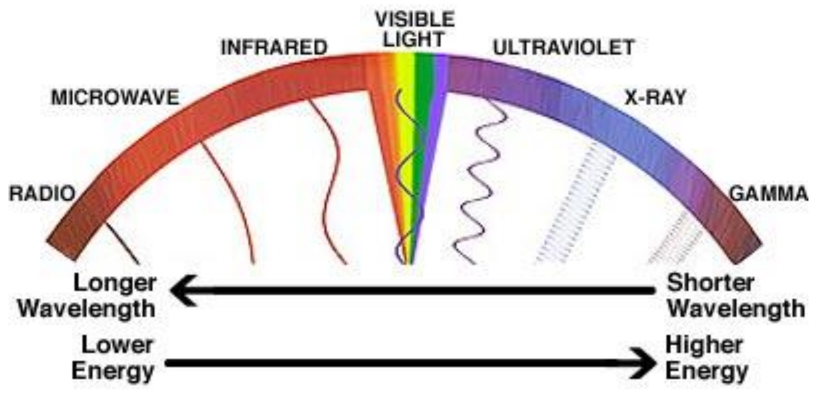
The purpose of this report is to document the steps and procedures taken in the design of the project, as well as all tests performed to validate the program conformance with the requirement. Refer to the following requirement text as originally provided by the university course for the project.

Requirement Text

Using the development board, program a chat protocol where two boards can communicate with each other via the IR transmitter and IR receiver. The text should be read from the terminal keyboard and transmitted to the other board where it will be received and displayed on its terminal and vice-versa. Any text transmitted should be displayed in blue and any text received should be displayed in green on the terminal.

Background

Infrared (IR) is commonly used as a wireless communication backend due to its invisible radiant energy properties. It is a type of electromagnetic radiation with a longer wavelength than normal visible light, thus the human eye cannot detect it.



The environment setup is listed in the table below.

Development Environment	
OS	`Windows 7`
IDE	`Keil uVision4`
Language	`C90`
Compiler	`arm-gcc`
Processor	`ARM Cortex-M4`
Board	`Stellaris LM4F120 (16MHz clock speed)`
Serial Port	`USB connection via Stellaris® Virtual Serial Port driver`
Serial Terminal	`Putty`
IR Components	`Transmitter: IR LED, Receiver: TSSP4038`

The Stellaris LM4F120 Launchpad is a low-cost evaluation platform for ARM® Cortex™-M4F-based microcontrollers from Texas Instruments. Stellaris features 16 General-purpose input/output (GPIO) pins on its board. GPIOs are a type of generic pins with input and output support that can be programmatically controlled. The IR LED and TSSP4038 are both connected to Stellaris via GPIO pins –pin C7 and pin C6 respectively. Stellaris also features a RGB LED along with two user switches.

The UART (universal asynchronous receiver/transmitter) subsystem is used for serial communications over a computer or peripheral device serial port. Stellaris does not have a serial port; however a virtual serial port can be used through its USB port. Therefore, any computer wishing to communicate with Stellaris via serial communication will be required to install drivers to support the virtual serial port.

Stellaris C API Interface

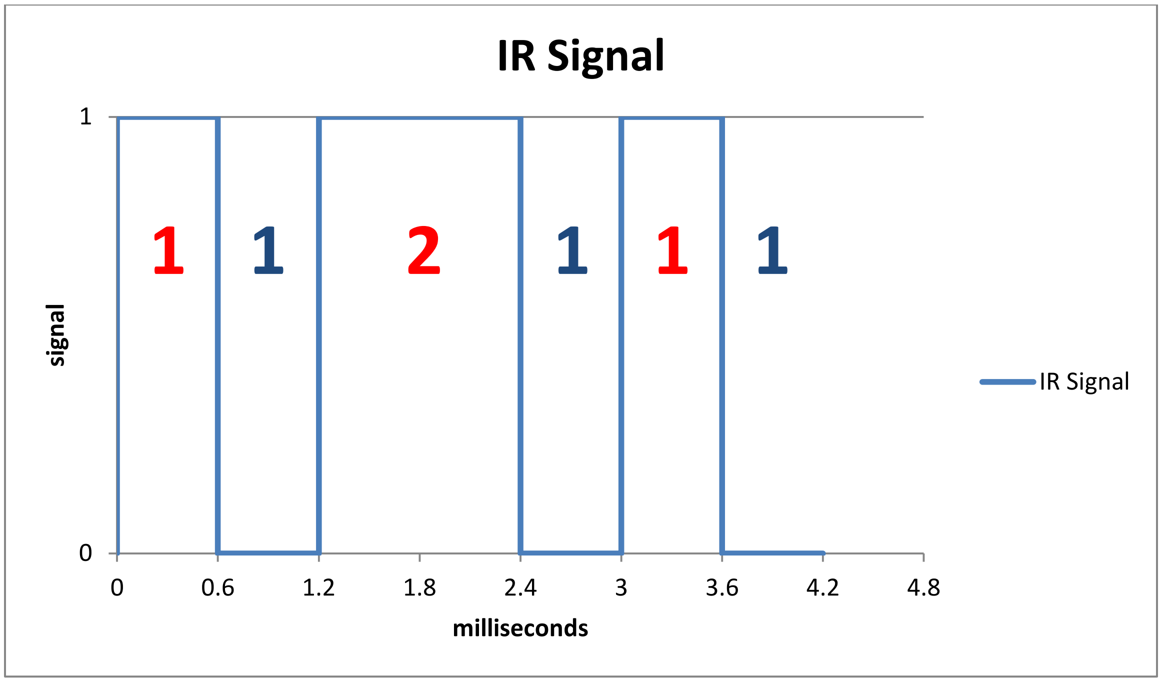
To interface with Stellaris' hardware functionalities, a small application programming interface (API) was developed for the project. It is contained in the `api.h` header file within the source folder. The following is an excerpt of the API header.

```
#ifndef API_H
#define API_H

/* ... */

// UARTs
#define uart_0                      0x4000C000

/* ... */

// UART APIs
#define uart_data(uart)             (*((uint32*) (uart + 0x000)))
#define uart_flag(uart)             (*((uint32*) (uart + 0x018)))
#define uart_ibrd(uart)             (*((uint32*) (uart + 0x024)))
#define uart_fbrd(uart)             (*((uint32*) (uart + 0x028)))
#define uart_lcrh(uart)             (*((uint32*) (uart + 0x02C)))
#define uart_ctrl(uart)             (*((uint32*) (uart + 0x030)))

/* ... */

#endif
```

The API is minimal, easy to use and offers no performance impact on the project. For example, to disable `uart_0`, a single line of `uart_ctrl(uart_0) &= ~0x1;` would suffice.

Protocol Design

The protocol is a defined specification on how to convert `messages` to IR signals and vice versa. A `message` is an arbitrary sized `byte` sequence, intended to be delivered from one point to the other point. The protocol uses pulse-width modulation (PWM) techniques to produce special patterns in the IR signals. By using these patterns, meaningful integers can be encoded in the pulse width of the signals. Consider the following graph for an example. By reading the graph, the integers carried in the signals are **1 – 1 – 2 – 1 – 1 – 1**.



`LOW` signals always have a fixed width and used as delimiters only. Therefore to interpret the `message` from the IR signals, only the integers in the `HIGH` signals are taken into consideration. So in this case, the integers being transmitted through this IR signal graph is read as **1 – 2 – 1**. The reason for doing so is to ease the signal sampling process, and also minimize the chance of error during decoding. This technique is also widely used in other IR applications, most notably TV remotes.

As illustrated in the graph above, the `pulse width unit` of each integer is `0.6 ms`. That is, an integer of 1 will cause the signal to last `0.6 ms`, whilst an integer of 2 will cause the signal to last `1.2 ms` and so on.

Opcodes

The next step in designing the protocol is to assign meanings to each of the integers being transmitted through IR signals. These meaningful integers are better known as `opcodes` (operation codes). For simplicity's sake, the protocol only defines a small number of `opcodes`. The table below lists all supported operations and their codes.

Opcode	Value	Description
`ops_0`	H1	Bit 0.
`ops_1`	H2	Bit 1.
`ops_beg`	H3	Starting opcode – all messages should start with this opcode.
`ops_end`	H4	Ending opcode – all messages should end with this opcode.
`ops_p`	L1	Pausing opcode – this is the `LOW` signal delimiter (as described previously) `opcode` sent in-between every other `opcode`.
`ops_count`	-	Pseudo opcode – the value represents the number of defined `opcodes`.
`ops_n`	-	Pseudo opcode – indicates no opcode is available. Implementation should ignore it and continue.

The `Value` column indicates the state of the signal and its duration. For example, a value of `H3` means the signal will be `HIGH` with an integer of `3`, and `L1` means the signal will be `LOW` with an integer of `1` etc.

A `message` can be constructed as the following graph. Notice that `ops_p` is inserted after every `opcode` except for itself acting as the delimiter (as described previously). The `message` begins with `ops_beg`, followed by an `ops_p`, finally ending with `ops_end` and `ops_p`.

`ops_beg`	**`ops_p`**	**`<< DATA >>`**	**`ops_end`**	**`ops_p`**

The omitted `DATA` section in the graph above is shown below.

`ops_1`	**`ops_p`**	**`ops_0`**	**`ops_p`**	**`ops_1`**	**`ops_p`**	**`ops_1`**	**`ops_p`**

Referring to the `opcode` table from before, `ops_0` and `ops_1` refers to `bit 0` and `bit 1` respectively. Therefore by reading the graph above, a 4-bit integer can be found. Bear in mind that the protocol uses a `LSB 0` numbering order – the least significant bit will be transmitted first and the most significant bit will be transmitted last.

<< Results >>	
Bits	1101
Size	4
Dec	13
Hex	D

With the `pulse width unit` configured at `0.6 ms`, the transmission speed varies between `69 bytes` to `104 bytes` per sec.

`0xFF`: (ops_1 x 8) + (ops_p x 8) = `14.4` ms per `byte` (69 `bytes` per sec)

`0x00`: (ops_0 x 8) + (ops_p x 8) = `9.6` ms per `byte` (104 `bytes` per sec)

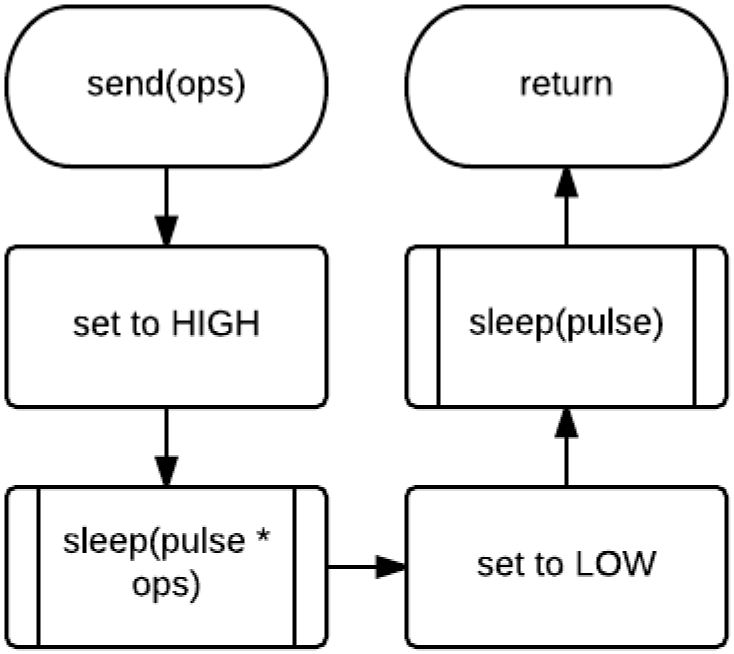
The following is the code declaration of the `opcodes`.

```
// filename: ops.h
typedef enum
{
	// ignore
	ops_n,

	// bit 0
	ops_0,

	// bit 1
	ops_1,

	// begin
	ops_beg,

	// end
	ops_end,

	// count
	ops_count,

	// pause
	ops_p
} ops;
```

The values of the `opcodes` are assigned sequentially based on their position in the declaration. Refer to the C90 Language Specification for more information on `enums`.

Encoding

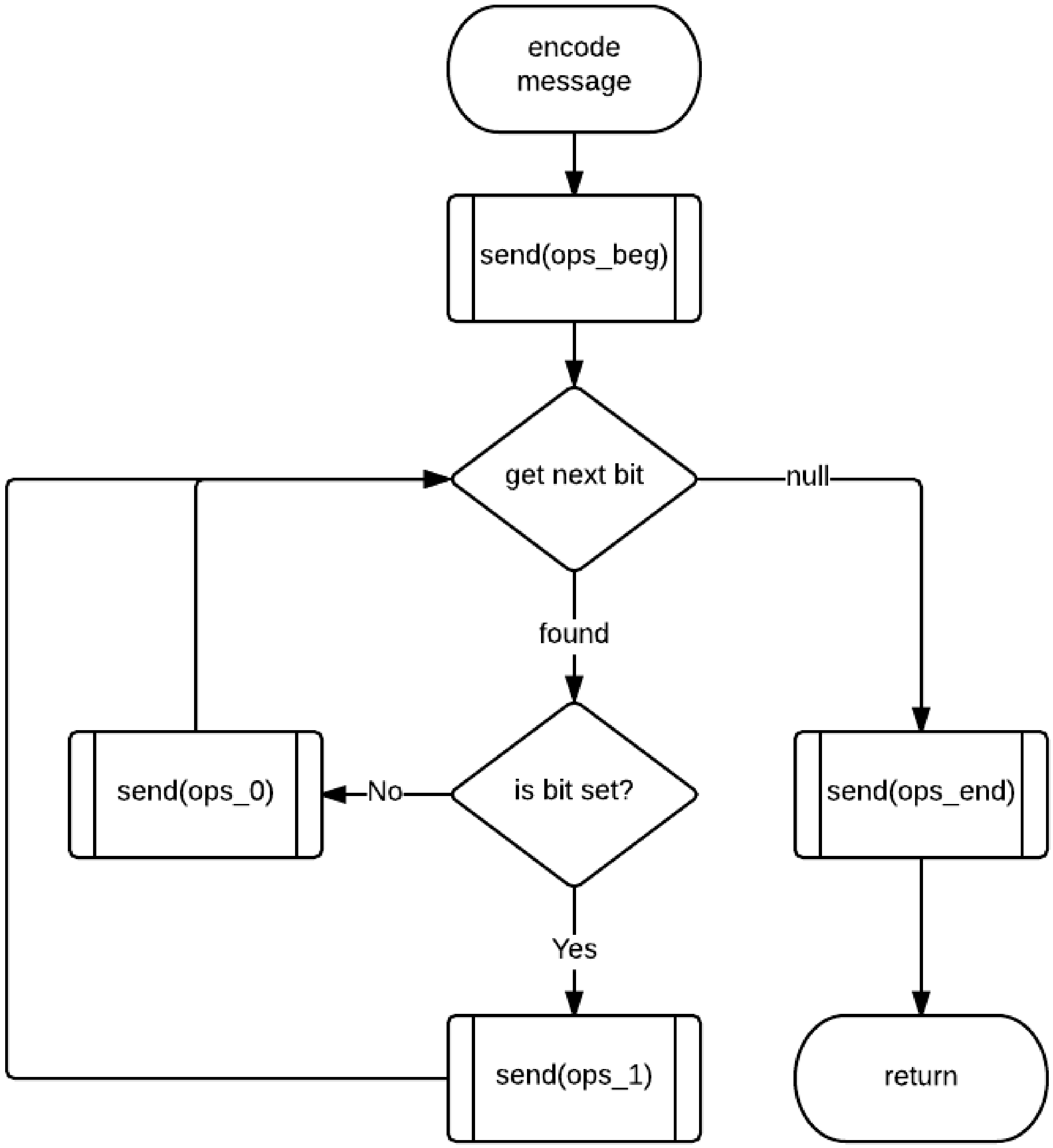
A function was created to transmit IR signals based on a given opcode. The logic behind it is illustrated in the following flowchart. In a summary, turn on the IR LED, and halt the processor for an amount of time based on the opcode's value. This will create the desired pattern in the IR signals.



The following is the code implementation of the flow chart. Just as the flowchart depicts, the IR LED pin is toggled to HIGH, and then the processor is halted for the amount of duration based on the opcode value.

```
// filename: ir.write.c
__inline static void write_ops(ops value)
{
        // HIGH
        gpio_data(port_c) |= ir_out;
        sleep(ir_pulse_us * value);

        // LOW
        gpio_data(port_c) &= ~ir_out;
        sleep(ir_pulse_us);
}
```

Now for the encoding function responsible for converting bytes into opcodes. The logic is documented in the following flowchart. Start by sending ops_beg, then iterate through each bit for each of the bytes and send ops_1 if bit is set else ops_0. Finally send an ops_end to signal end of transmission.

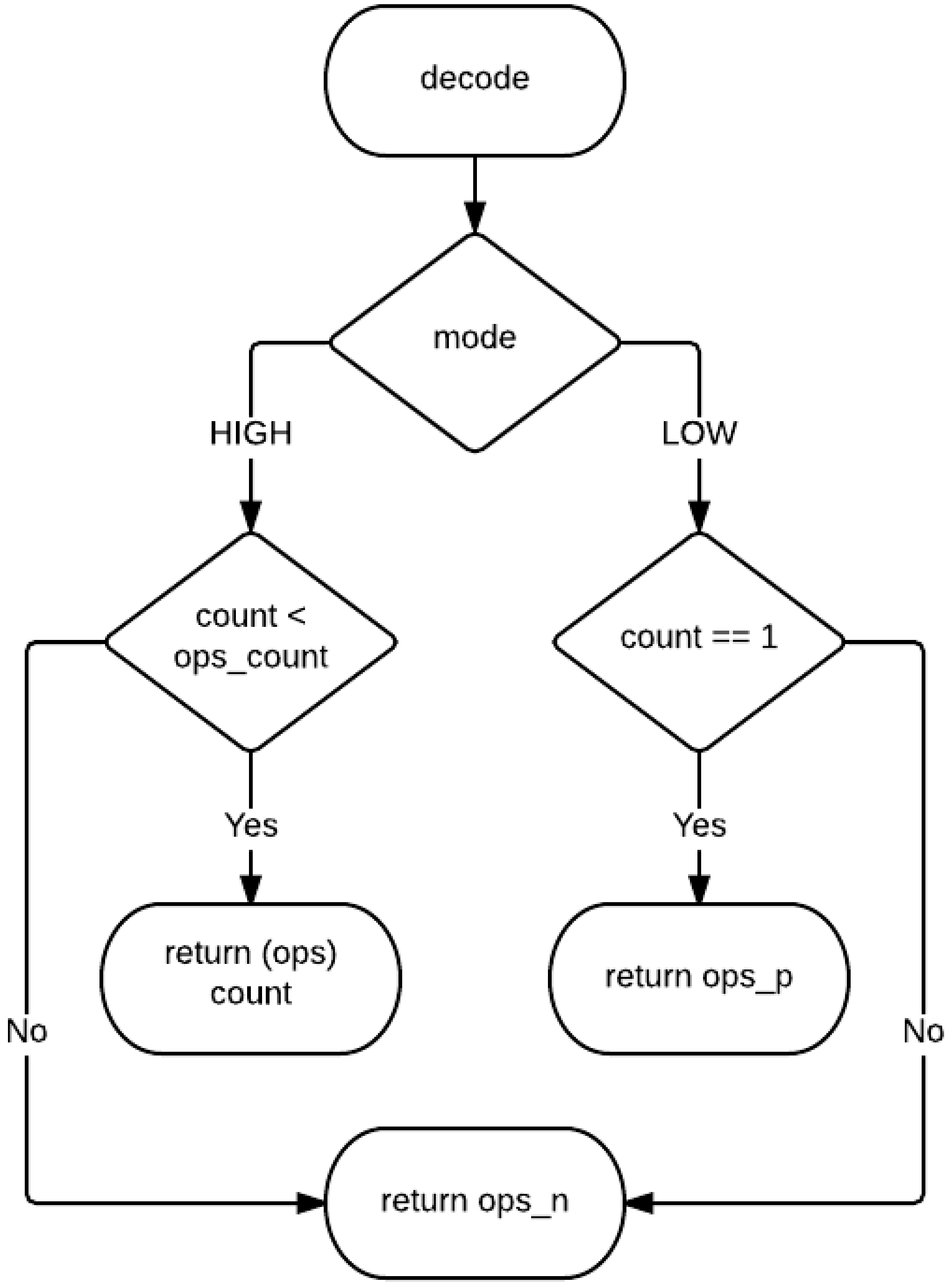


The following is the code implementation based on the flowchart above. Please refer to the source file `ir.write.c` for comments on each of the global variables being used here.

```c
// filename: ir.write.c
static void encode(void)
{
        uint32 i;

        // start
        write_ops(ops_beg);

        for (i = 0; i < index; i++)
        {
                uint32 b = 8, c = line[i];
                while (b--)
                {
                        // toggle
                        write_ops(c & 0x1 ? ops_1 : ops_0);
                        c >>= 1;
                }
        }

        // end
        write_ops(ops_end);
}
```

Decoding

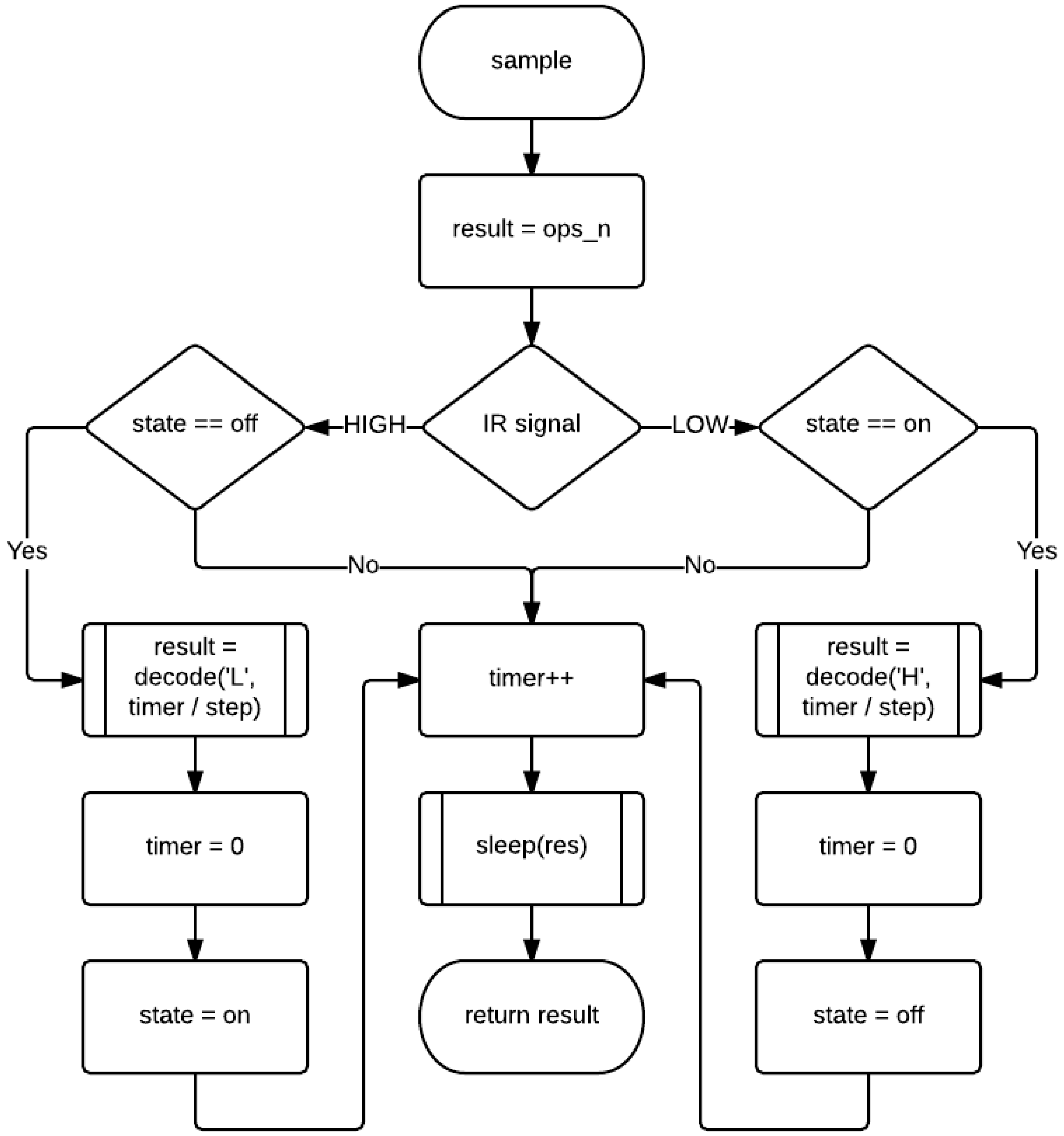
A function is created for converting signal pulses into `opcodes`. The logic is fairly simple, and depicted in the following flowchart. First check if the signal is `HIGH` and also falls within the correct number of `opcodes`, then simply cast the value into the type. Likewise, the same is done for `LOW` signal.



The following is the code implementation based on the flowchart above.

```
static ops decode(uint8 mode, uint32 count)
{
        switch (mode)
        {
                case 'H':
                        if (count < ops_count)
                        {
                                return (ops) count;
                        }
                        break;
                case 'L':
                        if (count == 1)
                        {
                                return ops_p;
                        }
                        break;
        }

        return ops_n;
}
```

The next step was to design a function to sample and record IR signals. Refer to the following flowchart detailing the necessary steps in achieving the task.



The code below is the implementation based on the flowchart.

```c
static ops sample(void)
{
        ops result = ops_n;

        if ((gpio_data(port_c) & ir_in))
        {
                if (state == ir_off)
                {
                        result = decode('L', timer / step);
                        timer = 0;
                        state = ir_on;
                }
        }
        else
        {
                if (state == ir_on)
                {
                        result = decode('H', timer / step);
                        timer = 0;
                        state = ir_off;
                }
        }

        timer++;
        sleep(ir_pulse_res);

        return result;
}
```

Program Design

A full-duplex channel was desired for the program's architecture. The biggest challenge in attempting to achieve such functionality was multi-tasking all the subsystems. Since a full-duplex channel would require the transmitter and the receiver to be active simultaneously, careful design was taken to ensure each task can be run smoothly and concurrently. One of the immediate concerns was whether it was possible with a 16MHz clock speed. However due to the time constraint and limited knowledge on the said fields, a half-duplex communication channel was opted for in the end.

Modules

As mentioned before, the project contains a single header file of Stellaris' hardware APIs. Multiple modules were then developed to consume this API. Each module is dedicated to a specific task of the project. For more information refer to the table below.

Name	Files	Purpose
API	`api.h, api.c`	Stellaris Hardware API
IR	`ir.h, ir.c, ir.read.c, ir.write.c`	Infrared Protocol
LED	`led.h, led.c`	Stellaris RGB LED
Main	`main.c`	Program entry point
Switch	`sw.h, sw.c`	Stellaris switch functions
UART	`uart.h, uart.c`	Stellaris UART 0 functions

Apart from the modules, there is a `Startup.s` file which is essentially responsible for initializing the C runtime and calling the entry point in main.c. Interrupt handlers are defined within the file.

Startup

Below is the main function of the program, which serves as the entry point. Upon starting up, the program initializes all required hardware functionalities by calling the `*_init` function of their respective modules.

```c
int main(void)
{
        // setup hardware
        uart_init();
        ir_init();

        led_init();
        led_rgb(led_green);

        // assign and reset mode
        ir_reset(mode = read_mode);

        // register sw1 press event
        sw_init(sw_one, change_mode);

        // execute mode indefinitely
        while (1) mode();
}
```

Running Modes

As mentioned before, the project is built with a half-duplex channel only. The program was split into two modes, each of their responsibilities are listed below:

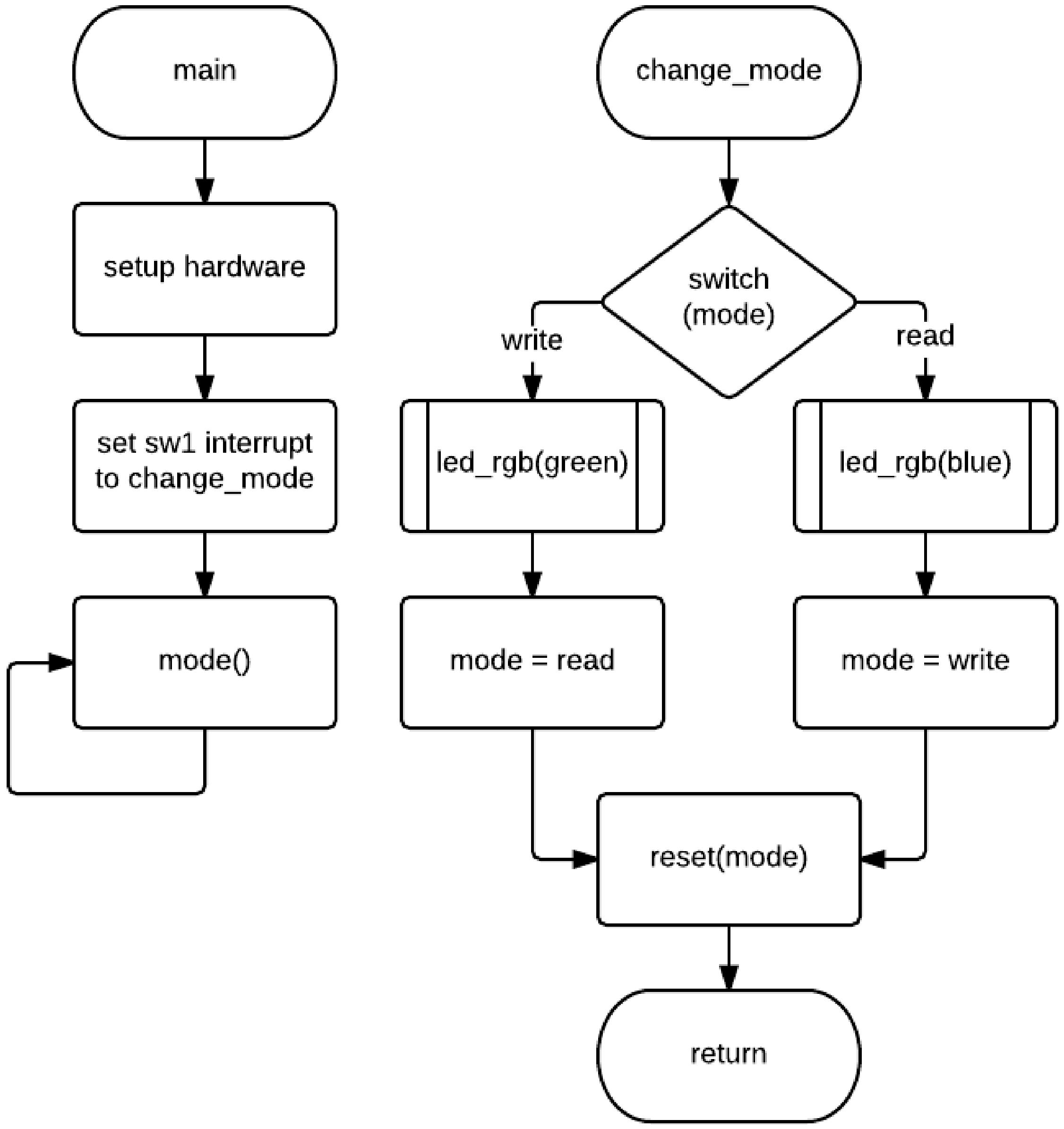
- Reading – sample IR signals, decode them into a `message`, and then printing it to the terminal
- Writing – read a `message` from the terminal, encode it into IR signals, and then transmit them

The modes can be switched by pressing the `sw1` button on the Stellaris® LM4F120.

```c
// called when sw1 is pressed
static void change_mode(void)
{
        // toggle mode and LED

        if (mode == write_mode)
        {
                led_rgb(led_green);
                mode = read_mode;
        }
        else if (mode == read_mode)
        {
                led_rgb(led_blue);
                mode = write_mode;
        }

        // reset mode
        ir_reset(mode);
}
```

While in **Read** mode, the on board RGB LED will light up as **GREEN**, whilst in **Write** mode the on board RGB LED will be **BLUE**. For more information on each of the modes, please refer to the source file `ir.read.c` and `ir.write.c` respectively.

Overview



Testing

This project only utilizes a small number of subsystems on the Stellaris. Hence the testing phase is relatively straightforward.

Subsystem	Testing Method
GPIO	Using Digital storage oscilloscope (DSO) to verify pulse amplitude and frequency.
UART	Using Putty as the terminal backend to send and receive data.
Interrupt	Verify the on board switches respond properly.

User Manual

1. Ensure everything is turned off and unplugged to prevent damage to the Stellaris boards
2. Setup hardware connection and position them appropriately
3. Connect both boards to each of their computers
4. Setup the serial terminal on each computer
5. Launch program

Upon starting up, both programs will be in **Read** mode. Toggle one of them into **Write** mode and begin usage.

Pin Connections

- **Pin C7** is for sending
- **Pin C6** is for reading

Wires can be used instead to replace the IR components, in which the program would still function in “wired” connection.

UART Configuration

Any serial communication supported terminal will suffice for this program. However it needs to be configured with the following settings.

- **Baud rate**: 115200
- **Data bits**: 8
- **Stop bits**: 1
- **Parity**: None
- **Flow Control**: None

References

Analog, Embedded Processing, Semiconductor Company, Texas Instruments - TI.com. (n.d.). *Analog, Embedded Processing, Semiconductor Company, Texas Instruments - TI.com*. Retrieved October 27, 2014, from http://www.ti.com

The Science Company® | Chemicals | Lab Supplies | Microscopes | pH Meters | Refractometers. (n.d.). *The Science Company® | Chemicals | Lab Supplies | Microscopes | pH Meters | Refractometers.* Retrieved October 27, 2014, from http://www.sciencecompany.com/Assets/images/EMSpectrum.jpg

Cortex-M3/M4F Instruction Set - TECHNICAL USER'S MANUAL. (2011). Austin, US: TEXAS INSTRUMENTS.

Cortex® -M4 Technical Reference Manual (r0p1 ed.). (2010). Cambridge, England: ARM.

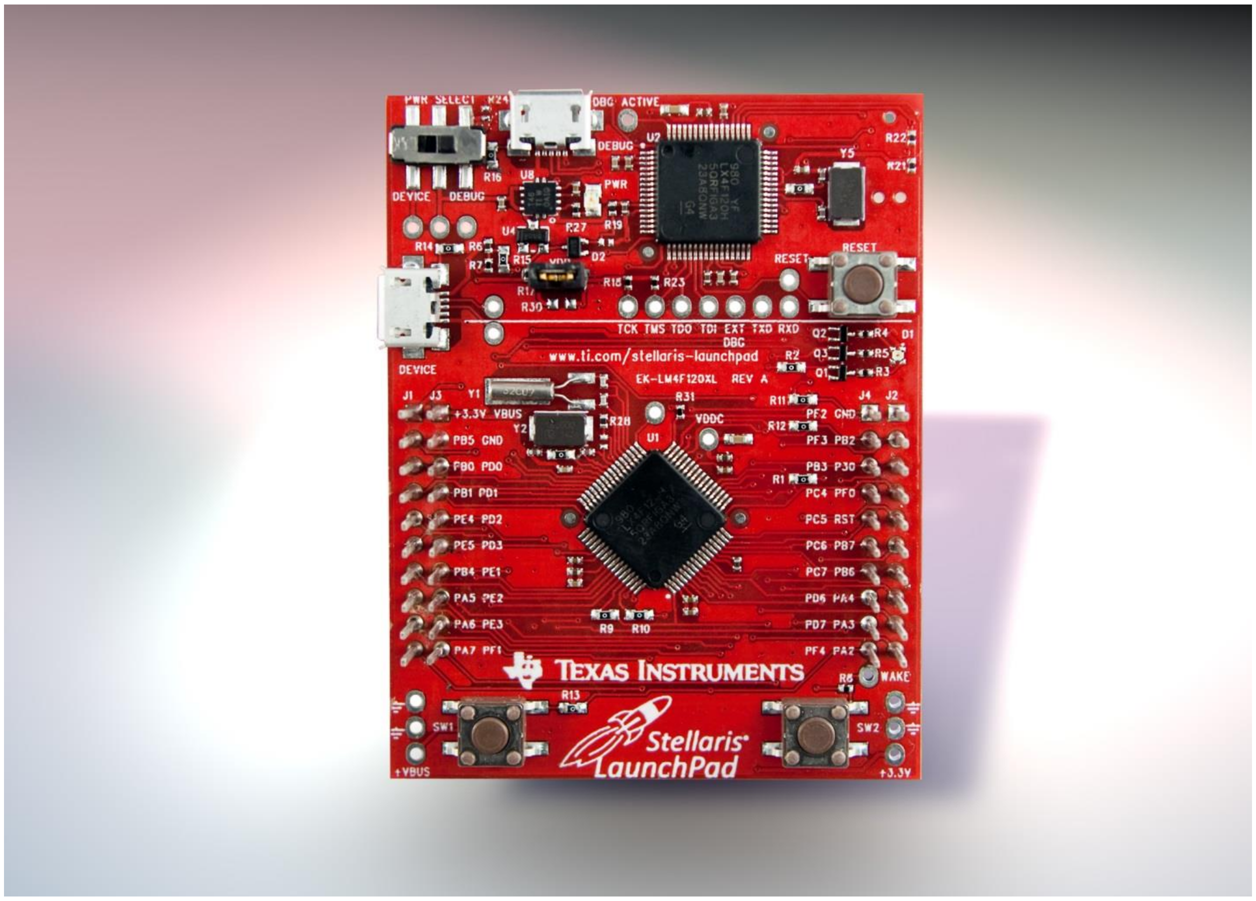
Stellaris® LM4F120 LaunchPad Evaluation Board User Manual. (2012). Austin, US: TEXAS INSTRUMENTS.

Stellaris® LM4F120H5QR Microcontroller DATA SHEET. (2012). Austin, US: TEXAS INSTRUMENTS.

Semiconductors. (n.d.). *Vishay*. Retrieved October 27, 2014, from http://www.vishay.com

Appendix

Stellaris Board



IR LED

IR Receiver (TSSP4038)