Bluetooth Trellis

Bluetooth trellis is a collection of libraries that make it easier to connect external hardware with a computer over a Bluetooth serial port via a PIC microcontroller.

It doesn't really make sense to have just one public API for this project. If anything, that API would be the code that runs on the PC side (libbluetrellis). The entire project is documented in this markdown file, but for grading purposes, I recommend that you focus on the libraries libi2c, libtrellis, and libuart as sources of our main public library functions.

Table of Contents

- PIC
  - libi2c
  - libtrellis
  - liblcd
  - libuart
  - libuarttrellis
- PC
  - libbluetoothserialport
  - libbluetrellis

PIC

On the PIC side, there are 5 relevant libraries: libi2c, libuart, libtrellis liblcd, and libuarttrellis.

This code (as a whole) and more generally libuarttrellis is designed to use the PIC as the controller for an RGB game pad that you can connect to over Bluetooth. All the other libraries are more general purpose.

The hardware used is as follows

- Adafruit NeoTrellis RGB Driver
- Adafruit Silicone Elastomer 4x4 Button Keypad
- DSD Tech HC-06 Bluetooth Module
- Sitronix ST7032 LCD

The DSD Tech website for the HC-06 is incredibly finicky, so the link may be broken by the time you read this. The Amazon link is better (but the data on it is inaccurate) Amazon DSD Tech HC-06

Libuarttrellis (the top level library) could be replaced with another library that uses a different set of hardware (maybe it interfaces with an led matrix or some other piece of hardware). This would be as simple as sending different commands with different structures over libi2c and libuart.

In general, the code is designed to be extensible and reusable.

libi2c

The libi2c library is designed to provide a unified method for sending and receiving data over an i2c bus with the I2C2 peripheral.

Functions

libi2c only provides three functions:

```
/**
 * Begins i2c communication with peripheral I2C2 at 100000 baud.
 */
void i2c_init(void);
```

```
/**
 * Executes a blocking read over I2C
 * @param i2c_addr The address of the thing to read.
 * @param prefix The I2C prefix (likely address bytes)
 * @param prefix_len The length of the prefix
 * @param dest The place to store the result of the read
 * @param size The size of the read buffer
 * @param delay The delay between the initialization command and the read.
 */
void i2c_read(uint8_t i2c_addr, const uint8_t *prefix, uint8_t prefix_len,
        uint8_t *dest, uint8_t size, int delay);
```

```
/**
 * Executes a blocking send over I2C
 * @param i2c_addr The address of the thing to send
 * @param prefix The I2C prefix (likely address bytes)
 * @param prefix_len The length of the prefix
 * @param data The data to send
 * @param data_len The length of the data
 */
void i2c_send(uint8_t i2c_addr, const uint8_t *prefix, uint8_t prefix_len,
        const uint8_t *data, uint8_t data_len);
```

Usage

Proper users of this library should begin by calling i2c_init() (which initializes i2c communication through the I2C2 peripheral). They can then call functions to read and send over that bus.

```
int main(void)
{
    i2c_init();

    // MANUALLY send a set_led command to the trellis
    const uint8_t write_prefix[2] = {
        SEESAW_NEOPIXEL_BASE, SEESAW_NEOPIXEL_BUF
    };
```

```
    const uint8_t write_data[5] = {
        0x00, 0x00, 0xFF, 0xFF, 0xFF
    };
    i2c_send(TRELLIS_ADDR, write_prefix, 2, write_data, 5);

    // MANUALLY read the number of key events in the trellis
    const uint8_t read_prefix[2] = {
        SEESAW_KEYPAD_BASE, SEESAW_KEYPAD_COUNT
    };
    uint8_t read_dest;
    i2c_read(TRELLIS_ADDR, read_prefix, 2, read_dest, 1, 500;

    while (1);

    return 0;
}
```

libtrellis

The libtrellis library is designed to control a single ADAFRUIT NeoTrellis with a PIC24. As Adafruit does not put out any documentation for their i2c protocol or hardware, all code for this library had to be reverse engineered from the Adafruit_Seesaw repository on GitHub.

This library makes use of the I2C2 and TMR3 peripherals.

Structs

This library provides a couple of structs to make interfacing with the trellis easier:

```
/**
 * A struct defining the layout of a key_event recieved from a keypad read.
 */
union key_event {
    struct {
        uint8_t edge : 2;
        uint8_t num : 6;
    };
    uint8_t raw;
};
```

```
/**
 * A struct that defines the layout of a key_state to be sent
 * in a set_keypad_events command.
 */
union key_state {
    struct {
        uint8_t state : 1;
        uint8_t active : 4;
        uint8_t extra : 3;
```

```
    };
    uint8_t raw;
};
```

Functions

The functions provided by this library are as follows:

```
/**
 * Sends the required initialization commands over the trellis. Also,
 * initializes a 20ms frame timer on TMR3.
 *
 * i2c_init() from libi2c must be called before this function is run.
 */
void trellis_init(void);
```

```
/**
 * Preforms a blocking delay to wait for the 20ms frame timer.
 * Resets the frame timer to zero when the function completes.
 */
void await_frame(void);
```

```
/**
 * Sets a keypad event to be tracked or ignored by the trellis.
 * @param num The number of the key (0-15)
 * @param edges The edge in question
 * @param active Whether or not it should be tracked.
 */
void set_keypad_event(uint8_t num, uint8_t edge, uint8_t active);
```

```
/**
  * Reads at most max_size button events into a buffer.
  * @param buffer The buffer to write into.
  * @param max_size The maximum number of key_events to read.
  * @return The actual number of key_events read.
  */
int get_button_events(union key_event *buffer, uint8_t max_size);
```

```
/**
 * Sends a set led command to the trellis.
 * @param num The led to set.
 * @param g The green value.
```

```
 * @param r The red value.
 * @param b The blue value.
 */
void set_led(uint8_t num, uint8_t g, uint8_t r, uint8_t b);
```

```
/**
 * Sends a set display command to the trellis.
 * @param colors An array of colors for each led in GRB format.
 */
void set_display(uint8_t colors[16][3]);
```

```
/**
 * Tells the trellis to update the neopixels with the new values that it has
 * in it's buffer. This function should be called when you want to see the
 * new data written by the set_led and set_display commands.
 */
void display_show(void);
```

```
/**
 * Sends the required initialization commands over the trellis. Also,
 * initializes a 20ms frame timer on TMR3. By default, tracks every rising
 * and falling edge.
 *
 * i2c_init() from libi2c must be called before this function is run.
 */
void trellis_init(void);
```

Usage

This library must be used with the libi2c library which must be initialized before use. Additionally, trellis_init(void) must be called before any other function in libtrellis can be called. Note that trellis_init(void) sets the button tracking events to rising and falling edges for all the buttons.

To initialize the trellis and set the display to blue, you might do something like this. It is important to note once again that colors are sent in GRB format as opposed to RGB format.

```
int main(void)
{
    const uint8_t colors[16][3] = {
        { 0x73, 0x00, 0xFF },
        { 0x73, 0x00, 0xFF },
        { 0x73, 0x00, 0xFF },
        { 0x73, 0x00, 0xFF },
        { 0x73, 0x00, 0xFF },
```

```
        { 0x73, 0x00, 0xFF },
        { 0x73, 0x00, 0xFF },
        { 0x73, 0x00, 0xFF },
        { 0x73, 0x00, 0xFF },
        { 0x73, 0x00, 0xFF },
        { 0x73, 0x00, 0xFF },
        { 0x73, 0x00, 0xFF },
        { 0x73, 0x00, 0xFF },
        { 0x73, 0x00, 0xFF },
        { 0x73, 0x00, 0xFF },
        { 0x73, 0x00, 0xFF }
    };

    i2c_init();
    trellis_init();
    set_display(colors);

    while (1);

    return 0;
}
```

Below is another example that both initializes the trellis and begins reading button presses from it to light up the LEDs. Before it does this, it stops tracking the button presses from the 15th button (row 4, column 4).

```
int main(void)
{
    const uint8_t blue[3] = { 0x73, 0x00, 0xFF };
    const uint8_t red[3]  = { 0x00, 0xFF, 0x00 };
    const uint8_t colors[16][3] = {
        { 0x73, 0x00, 0xFF },
        { 0x73, 0x00, 0xFF },
        { 0x73, 0x00, 0xFF },
        { 0x73, 0x00, 0xFF },
        { 0x73, 0x00, 0xFF },
        { 0x73, 0x00, 0xFF },
        { 0x73, 0x00, 0xFF },
        { 0x73, 0x00, 0xFF },
        { 0x73, 0x00, 0xFF },
        { 0x73, 0x00, 0xFF },
        { 0x73, 0x00, 0xFF },
        { 0x73, 0x00, 0xFF },
        { 0x73, 0x00, 0xFF },
        { 0x73, 0x00, 0xFF },
        { 0x73, 0x00, 0xFF },
        { 0x73, 0x00, 0xFF }
    };

    // Initialize the display.
    i2c_init();
    trellis_init();
```

```
    set_display(colors);

    // Deactivate the 15th key.
    set_keypad_event(15, EDGE_RISING, 0);
    set_keypad_event(15, EDGE_FALLING, 0);

    while (1) {
        union key_event events[30];
        union key_event *key = events;
        struct button_event button;
        int num_events;

        // Wait for the next frame and get button events.
        await_frame();
        display_show();
        num_events = get_button_events(events, 30);

        // Handle the events.
        while (num_events--) {
            if (events[num_events].edge == EDGE_RISING) {
                set_led(key->num, blue[0], blue[1], blue[2]);
            } else {
                set_led(key->num, red[0], red[1], red[2]);
            }
        }
    }

    return 0;
}
```

liblcd

The liblcd library is designed to provide a simple way to initialize and send commands to a Sitronix ST7032 LCD.

Functions

The functions provided by this library are as follows:

```
/**
 * Sends an lcd command over i2c.
 * @param pkg The payload for the command.
 */
void lcd_cmd(uint8_t pkg);
```

```
/**
 * Initializes the lcd. Must not be called before i2c_init() from libi2c.
 */
void lcd_init(void);
```

```
/**
 * Sends a lcd set cursor command over i2c.
 * @param row The new row of the cursor.
 * @param col The new column of the cursor.
 */
void lcd_set_cursor(uint8_t row, uint8_t col);
```

```
/**
 * Sends a putc command to the lcd (sets a single character)
 * @param c The character to put to the buffer.
 */
void lcd_putc(uint8_t c);
```

```
/**
 * Sends a puts command to the lcd (sets a full string)
 * @param string The string to be drawn.
 */
void lcd_puts(uint8_t string[]);
```

Usage

This library must be used with the library libi2c. Before you can send any commands to the lcd, you must initialize both the i2c bus (with i2c_init()) and the display itself (with lcd_init()).

If you wanted to send a string to the display to start at row zero, column zero, that would look something like this:

```
int main()
{
    i2c_init();
    lcd_init();
    lcd_set_cursor(0, 0);
    lcd_puts("Hello!");

    while (1);

    return 0;
}
```

If you wanted to send more data than that, all you would simply have to chain these commands together.

```c
int main()
{
    i2c_init();
    lcd_init();
    lcd_set_cursor(0, 0);
    lcd_puts("Hello!");
    lcd_set_cursor(1, 7);
    lcd_putc('a');

    while (1);

    return 0;
}
```

libuart

The libuart library is designed to simplify sending and recieving of commands over a UART bridge. In this project, we connect this UART bridge to a HC-06 UART to Bluetooth converter and connect our device to a PC. This library is not specific to that application however, and can be used on its own.

Functions

The functions provided by this library are as follows:

```c
/**
 * Initializes UART communication on pins RB7 and RB8 at 38400 baud.
 */
void uart_init(void);
```

```c
/**
 * Checks if the receive buffer is empty.
 * @return 1 if the buffer has contents.
 */
int uart_empty(void);
```

```c
/**
 * Send 'bytes' bytes of data over UART, preceeded by a command header.
 * @param header The header byte of the command.
 * @param data The address of the data to be sent.
 * @param bytes The number of data bytes in the command.
 */
void send_command(unsigned char header, unsigned char *data, unsigned char bytes);
```

```
/**
 * Consumes and returns the header byte of the top command in the queue.
 * Blocks if there is no data in the UART buffer.
 * @return The header byte of the current command.
 */
unsigned char get_command_header();
```

```
/**
 * Unpacks 'bytes' bytes of data into a command data array.
 * Blocks until enough bytes are read from the UART buffer.
 * @param com The address of a command where the data should be inserted.
 * @param bytes The number of bytes to take from the buffer.
 */
void get_command_body(unsigned char *dest, unsigned char bytes);
```

Usage

Before this library can be used, you must call the initialization function uart_init().

After this is done, you can immediately begin sending and receiving data. Sending data with this library is relatively easy. You must define a command consisting of a header character and a body then send it as a series of bytes with the function send_command().

The example below sends a single button press over UART. First we define the protocol (which includes commands 'A' and 'B'). Then we send one command over UART.

```
#define HEADER_A 'A'
#define HEADER_B 'B'

struct command_a {
    uint16_t int_data;
    char char_data;
};

struct command_b {
    uint32_t int_data;
    float float_data;
};

int main()
{
    struct command_a com_a;

    uart_init();

    // Send the first command.
    com_a.is_rising = false;
    com_a.button_num = 0;
```

```
    send_command(
        HEADER_A,
        (uint8_t *)com_a,
        sizeof(struct command_a)
    );

    while (1);

    return 0;
}
```

Getting command input is not much different, this second example extends the above example to mirror commands that it receives after sending the first button event.

```
#define HEADER_A 'A'
#define HEADER_B 'B'

struct command_a {
    uint16_t int_data;
    char char_data;
};

struct command_b {
    uint32_t int_data;
    float float_data;
};

void handle_a(void)
{
    struct command_a command;
    get_command_body((uint8_t *)command, sizeof(struct command_a));

    // Just resend the command over UART
    send_command(
        HEADER_A,
        (uint8_t *)command,
        sizeof(struct command_a)
    );
}

void handle_b(void)
{
    struct command_b command;
    get_command_body((uint8_t *)command, sizeof(struct command_b));

    // Just resend the command over UART
    send_command(
        HEADER_B,
        (uint8_t *)command,
        sizeof(struct command_b)
    );
```

```
}

void process_header(char header)
{
    switch (header) {
        case HEADER_A:
            handle_a();
            break;
        case HEADER_B:
            handle_b();
            break;
        default:
            break;
    }
}

int main()
{
    struct command_a com_a;

    uart_init();

    // Send the first command
    com_a.is_rising = false;
    com_a.button_num = 0;
    send_command(
        HEADER_A,
        (uint8_t *)com_a,
        sizeof(struct command_a)
    );

    while (1) {
        // Look for a valid command header
        char header = get_command_header();
        if (header != NULL) {
            // Call the handling function if we recieved a non-null header.
            process_header(header);
        }
    }

    return 0;
}
```

libuarttrellis

Libuarttrellis is a rather simple library. As was stated above, it wraps up libtrellis, liblcd, and libuart into a single library that makes the PIC function as a bluetooth, rgb gamepad.

Functions

The functions are defined as follows.

```
/**
 * Initializes i2c communication with the I2C2 peripheral as well as 38400
 * baud UART communication with peripheral U1 with TX on RB7 and RX on RB8.
 * Also sends i2c initializing commands to the LCD and trellis over i2c.
 * Initializes TMR3 as a frame timer for the trellis.
 */
void bluetrellis_init(void);
```

```
/**
 * Polls for button presses and sends a draw/show command to the display.
 * This function runs at a period of 20ms or greater. If 20ms has not
 * passed since the last call, the function will perform a blocking delay
 * until it has. Sends all button events over UART immediately.
 */
void poll_and_update(void);
```

```
/**
 * Handles commands sent over UART from the bluetooth device.
 */
void process_uart(void);
```

Usage

There is really only one proper way to use this library. It is shown in bluetrellis_main.c

```
int main(void) {
    setup();

    bluetrellis_init();

    while (1) {
        poll_and_update();
        process_uart();
    }

    return 0;
}
```

First the library is initialized with the bluetrellis_init() function. Then the library repeatedly calls poll_and_update() and process_uart(). This allows the PIC to handle incoming commands and forward button events over UART.

PC

On the PC side, there is are two libraries: libbluetoothserialport and libbluetrellis.

Libbluetoothserialport is an extarnal library developed by Agamnenztar that allows for multiplatform bluetooth serial port communication. This allows us to write bluetooth code that works for any operating system.

Libbluetrellis (the PC equivalent of libuarttrellis on the PIC) serves as a wrapper around libbluetoothserialport and provides a set of functions that allow the user to interface with the PIC acting as an rgb gamepad.

libbluetoothserialport

The documentation for libbluetoothserial port is contained on the original github page by Agamnentzar. bluetooth-serial-port

libbluetrellis

Libbluetrellis, as was stated above is designed to be a wrapper around libbluetoothserialport. The library exposes a series of functions that send commands over the bluetooth bridge.

As was stated above, this library could be replaced or extended to allow for interfacing with different hardware connected to the pic.

The library can be compiled with the option BLUE_STDIO=ON to rerout UART data to STDIO. This forces the compiler to define BLUE_STDIO before compiling the library hence the number of #ifndef's and #endif's in the source code. To learn more about building look at README.md in the build directory.

Command Descriptions

The library defines the following commands (these correspond with the libuarttrellis library on the PIC side)

```
enum {
    BUTTON_EVENT_HEADER = 'A',
    SHOW_HEADER = 'C',
    SET_LED_HEADER = 'D',
    SET_DISPLAY_HEADER = 'E',
    SET_LCD_HEADER = 'F',
};
```

```
/**
 * A struct that defines the layout of a button_event
 */
union button_event {
    struct {
        uint8_t button_num : 7;
        uint8_t is_rising : 1;
    };
    uint8_t raw;
};
```

```
/**
 * A struct that defines the layout of a set_led command
 */
union set_led {
    struct {
        uint8_t led_num;
        uint8_t color[3]; // RGB values for the led.
    };
    uint8_t raw;
};
```

```
/**
 * A struct that defines the layout of a set_display command
 */
union set_display {
    struct {
        uint8_t colors[16][3]; // Array of 16 GRB values.
    };
    uint8_t raw;
};
```

```
/**
 * A struct that defines the layout of a set_lcd comamnd
 */
union set_lcd {
    struct {
        uint8_t data[2][8]; // Array of strings for top and bottom row.
    };
    uint8_t raw;
};
```

Functions

The following functions are provided to simplify sending and recieving commands

```
/**
 * Constructs a blue_trellis object
 * @param addr a string containing the address of the bluetooth device
 */
blue_trellis(std::string addr);
```

```
/**
 * Sends a set_led command over bluetooth.
 * @param num The number of the led to set on the button pad
 * @param g The green value
 * @param r The red value
 * @param b The blue value
 */
void send_set_led(uint8_t num, uint8_t g, uint8_t r, uint8_t b);
```

```
/**
 * Sends a set_display command over bluetooth
 * @param colors An array of 16 GRB encoded colors to be displayed
 */
void send_set_display(const uint8_t colors[16][3]);
```

```
/**
 * Sends a set_lcd command over bluetooth
 * @param data The array of characters to be written to the lcd
 */
void send_set_lcd(const uint8_t data[2][8]);
```

```
/**
 * Non-blocking function that polls for a command header.
 * @return The header recieved (otherwise NULL)
 */
char poll_header();
```

```
/**
 * Performs a blocking wait for the body of a button_event
 * @return A struct containing the button event data.
 */
union button_event get_button_event();
```

```
/**
 * Destroys a blue_trellis object
 */
~blue_trellis();
```

Usage

Before this library can be used, one must simply construct an object. The constructor for this class takes one parameter which is the address.

```
blue_trellis bt = blue_trellis(bluetooth_addr);
```

Sending commands using this library is relatively self-explanatory. Calling different send functions queues up commands to be sent over the Bluetooth that will be sent and handled by the operating system.

The example below sends commands that reset the trellis display and LCD.

```
int main()
{
    const uint8_t colors[16][3] = {
        { 0x73, 0x00, 0xFF },
        { 0x73, 0x00, 0xFF },
        { 0x73, 0x00, 0xFF },
        { 0x73, 0x00, 0xFF },
        { 0x73, 0x00, 0xFF },
        { 0x73, 0x00, 0xFF },
        { 0x73, 0x00, 0xFF },
        { 0x73, 0x00, 0xFF },
        { 0x73, 0x00, 0xFF },
        { 0x73, 0x00, 0xFF },
        { 0x73, 0x00, 0xFF },
        { 0x73, 0x00, 0xFF },
        { 0x73, 0x00, 0xFF },
        { 0x73, 0x00, 0xFF },
        { 0x73, 0x00, 0xFF },
        { 0x73, 0x00, 0xFF }
    };

    const uint8_t init_lcd[2][8] = {
        { 'H', 'e', 'l', 'l', 'o', '!', ' ', ' ' },
        { ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ' }
    };

    blue_trellis bt = blue_trellis(bluetooth_addr);

    bt.send_set_display(colors);
    bt.send_set_lcd(init_lcd);

    return 0;
}
```

The second example extends the above example to read button events from the trellis and send corresponding set_led commands back (just like we did with the libtrellis example).

```

```

```
int main()
{
    const uint8_t blue[3] = { 0x73, 0x00, 0xFF };
    const uint8_t red[3]  = { 0x00, 0xFF, 0x00 };

    const uint8_t colors[16][3] = {
        { 0x73, 0x00, 0xFF },
        { 0x73, 0x00, 0xFF },
        { 0x73, 0x00, 0xFF },
        { 0x73, 0x00, 0xFF },
        { 0x73, 0x00, 0xFF },
        { 0x73, 0x00, 0xFF },
        { 0x73, 0x00, 0xFF },
        { 0x73, 0x00, 0xFF },
        { 0x73, 0x00, 0xFF },
        { 0x73, 0x00, 0xFF },
        { 0x73, 0x00, 0xFF },
        { 0x73, 0x00, 0xFF },
        { 0x73, 0x00, 0xFF },
        { 0x73, 0x00, 0xFF },
        { 0x73, 0x00, 0xFF },
        { 0x73, 0x00, 0xFF }
    };

    const uint8_t init_lcd[2][8] = {
        { 'H', 'e', 'l', 'l', 'o', '!', ' ', ' ' },
        { ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ' }
    };

    blue_trellis bt = blue_trellis(bluetooth_addr);


    bt.send_set_display(colors);
    bt.send_set_lcd(init_lcd);

    while (1) {
        struct button_event event;
        char header = bt.poll_header();
        if (header == blue_trellis::BUTTON_EVENT_HEADER) {
            event = bt.get_button_event();
        }
    }

    return 0;
}
```