Bluetooth Controller Project

Manuel Saldana

Winter 2020 - Summer 2020

Contents

1 Introduction

With its built-in Bluetooth capabilities, the BeagleBone Blue has the potential to interface with a variety of wireless devices. One such device includes wireless video game controllers, which could be used as a means to control robots built with the BeagleBone Blue. The purpose of this project was to explore this potential and find a way in which a user might be able to use a wireless video game controller with their BeagleBone Blue. This document explores one method a user can follow to read input from a video game controller in C code for use in various robotics projects.

This document pertains to the BeagleBone Blue and will focus only on the DualShock 4 controller for the PlayStation 4 and the Xbox One S controller. Both of these controllers use Bluetooth to connect to their respective consoles as well as a variety of other devices such as smartphones and computers. The remainder of this section will cover the basics of how to use these controllers.

1.1 The DualShock 4 Controller for the PlayStation 4

Figure 1: DualShock 4 Controller

Power On/Off: The DualShock 4 controller can be powered on by holding the PlayStation logo (PS) button on the front of the controller. When the controller is on, the light bar at the top of the controller will be lit. To turn off the controller, hold down the PS button until the light bar is switched off.

Pairing Mode: To enter pairing mode, hold the PS and SHARE buttons at the same time until the light bar at the top of the controller begins to blink white. When the controller is paired to a device, it will stop blinking and remain lit. By default, the light bar will be a solid blue color.

Charging: The DualShock 4 controller can be charged using a USB Micro B charging cable. Charging is indicated by the light bar at the top of the controller, which will pulse a yellow light. When charging is complete the light bar will turn off.

1.2 The Xbox One S Controller

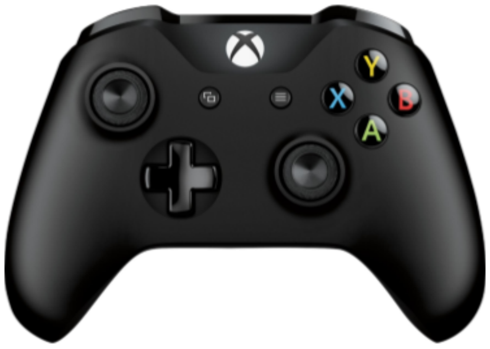


Figure 2: Xbox One S Controller

Power On/Off: The Xbox One S controller can be powered on by holding the Xbox logo (XBOX) button on the front of the controller. When the controller is on, the XBOX button will be lit. To turn off the controller, hold down the XBOX button until the light is switched off.

Pairing Mode: To enter pairing mode, press the CONNECT button at the top of the controller, between the right trigger (RT) and left trigger (LT) buttons. The XBOX button will begin to blink white. When the controller is paired to a device, it will stop blinking and remain lit.

Charging: The Xbox One S controller requires two AA batteries. Charging via USB cable is not available.

Bluetooth Capability: It is important to note that not all Xbox One controllers have Bluetooth capabilities. Only the Xbox One S controller, which is the latest Xbox controller, has Bluetooth capabilities. An Xbox One S controller can be distinguished by the plastic surrounding the XBOX button. If the plastic surrounding the XBOX button is the same as the plastic on the front face of the controller, then it is an Xbox One S controller.

2 The Joystick Package

The focus of this section will be to guide the user through the process of installing the **Joystick package** on their BeagleBone Blue. The Joystick package is essential to allowing the user to read input from a video game controller using C code.

While the Joystick package can be installed directly, it will be best for users to install the **Joystick Testing and Configuration Tool package**. This package is dependent on the Joystick package, so installing this package will also install the Joystick package.

2.1 Connecting to Wifi on the BeagleBone Blue

Installing packages requires an internet connection. The following steps will guide the user through the process of connecting to a WiFi network on the BeagleBone Blue using the **connmanctl** network manager.

Start the network manager by entering the following command:

```
connmanctl
```

Enable WiFi and scan for networks:

```
enable wifi
scan wifi
```

Then, turn on the agent:

```
agent on
```

Afterwards, enter the following command:

```
services
```

This will display a list of WiFi networks that are available. Each WiFi network will have a respective address listed beside it. To connect to a specific network, enter the command:

```
connect wifi_addressfromlist
```

If the network is password protected a prompt will appear asking the user to input a password. Once the correct password has been entered a confirmation message will appear notifying the user that the network has been connected to.

Finally, to exit the network manager:

```
exit
```

2.2 Installing the Joystick Package via the Joystick Testing and Configuration Tool

This next step will guide the user through the process of installing the Joystick Testing and Configuration Tool package. Prior to installing any package, the package lists for Linux must be updated. This can be achieved using the command:

```
sudo apt-get update
```

The following message will appear confirming that the update has been completed:

```
Reading package lists... Done
```

The package can now be installed. To install the package:

```
sudo apt install jstest-gtk
```

If successful, this package and all its dependencies should be installed.

2.3 Checking if a Package is Installed on Linux

This final step will make sure that both packages have been installed on the BeagleBone Blue. To check if a package is installed on Linux enter the command:

```
dpkg -s packagename
```

The following message will be displayed if a package has not been installed:

```
dpkg-query: package 'packagename' is not installed and no information is available
```

To check if the Joystick and Joystick Testing and Configuration Tool packages have been installed:

```
dpkg -s joystick
dpkg -s jstest-gtk
```

3 Bluetooth

The focus of this section will be to guide the user through the process of connecting a Bluetooth device to the BeagleBone Blue.

3.1 First Time Setup

Connecting an Xbox One S controller requires an **additional step** before the actual process can begin. Prior to using the **Bluetooth configuration tool**, the following command must be executed:

```
sudo sh -c 'echo 1 > /sys/module/bluetooth/parameters/disable_ertm'
```

This command disables the "Enhanced Re-Transmission Mode" Bluetooth protocol which is known to cause connectivity issues with Xbox One S controllers. If the BeagleBone is rebooted, this protocol will be reset. Therefore, this command will need to be executed **every time** an Xbox One S controller is to be connected via Bluetooth.

The following steps will guide the user through the process of connecting a video game controller to the BeagleBone Blue via Bluetooth for the first time. It will use the Bluetooth configuration tool, which should already be installed on the BeagleBone Blue. These steps will apply to both the DualShock 4 and Xbox One S controllers.

Start the Bluetooth configuration tool by entering the command:

```
bluetoothctl
```

Next, turn on the agent and set it as default:

```
agent on
default-agent
```

Power on the Bluetooth and set it as discoverable and pairable:

```
power on
discoverable on
pairable on
```

Power on the video game controller and set it to pairing mode. Refer to the Introduction section for instructions on how to do this. Afterwards, enter the command:

```
devices
```

This will display a list of devices that are discoverable and available for pairing. Both the DualShock 4 and Xbox One S controllers will be listed as "wireless controllers." Every device listed will have a respective MAC address. To pair the BeagleBone Blue to a specific device, enter the command:

```
pair devicemacaddress
```

A message will then appear asking to authorize the service. Enter `yes` to continue.

Finally, trust the device by entering the command:

```
trust devicemacaddress
```

The video game controller should now be paired. The DualShock 4 controller will stop blinking and remain lit. By default, the light bar will be a solid blue color. Likewise, the Xbox One S controller will stop blinking and remain lit. If this is true, the Bluetooth configuration tool can be exited using the following command:

```
exit
```

3.2 Connecting a Device to the BeagleBone Blue

The following steps will guide the user through the process of re-connecting a video game controller to the BeagleBone Blue after a successful first time setup. As previously mentioned, connecting an Xbox One S controller will require the following command to be executed:

```
sudo sh -c ‘echo 1 > /sys/module/bluetooth/parameters/disable_ertm’
```

Power on the video game controller and set it to pairing mode. Start the Bluetooth configuration tool:

```
bluetoothctl
```

On startup, devices that have been previously connected should appear in a list with their respective MAC address. If so, this will make connecting easier. If not, enter the following command for a list of available devices:

```
devices
```

Connect to a device using the provided MAC address. Recall that the TAB key can be used to auto-complete the device MAC address as it appears on the list.

```
connect devicemacaddress
```

The video game controller should now be connected to the BeagleBone Blue. Exit the Bluetooth configuration tool using the `exit` command.

4 Testing a Video Game Controller

The focus of this section will be to introduce the user to the Joystick Testing and Configuration Tool, as well as introduce the user to the axes and buttons available on the DualShock 4 and Xbox One S controllers.

4.1 Using the Joystick Testing and Configuration Tool

The Joystick Testing and Configuration Tool is a package that is very helpful for making sure a video game controller is properly connected. It is also helpful in determining the number of available axes and buttons on a controller, as well as which number each axis and button corresponds to.

It is important to note that the BeagleBone Blue recognizes both the DualShock 4 and Xbox One S controllers as **joystick input devices**. No matter which controller is being used, the file path is as follows:

```
/dev/input/js0
```

This path will be important later on when reading input from a controller using C code. For now, this path will be used for testing a controller. With both the video game controller connected and the packages installed, begin the tool by entering the following command:

```
jstest /dev/input/js0
```

A list of all available axes and buttons, each with their own corresponding number and value/state, will be displayed in the terminal. Changes to these values/states will be displayed on new lines. Axes will range in values from `-32767` to `32767` and buttons will be in either an `on` or `off` state. To exit the tool use `CTRL + C`.

4.2 Mappings for the DualShock 4 and Xbox One S Controllers

The last page of this document contains diagrams for the controller mappings of both the DualShock 4 and Xbox One S controllers.

The DualShock 4 controller has 8 axes and 13 buttons. The touch pad and touch pad buttons are not available.

The Xbox One S controller has 8 axes and 15 buttons. BUTTON 2 and BUTTON 5 do not exist. The VIEW button and XBOX button are not available

5 C Programs Using the Joystick Package

The focus of this section will be to introduce the user to the C code provided on the GitHub page.

5.1 GitHub and the Joystick API Documentation

The GitHub repository for this project, which includes the code presented in this section as well as the controller mappings shown in the previous section, can be found by following the URL: `https://github.com/MSaldana362/bluetoothcontroller`.

The Joystick API documentation, which was referred to for a majority of the code presented in this section, can be found by following the URL: `https://www.kernel.org/doc/Documentation/input/joystick-api.txt`.

It is highly recommended for new users to look over the code to gain a better understanding of how the programs work. The code has been heavily commented with the hope that this will provide the user with enough insight to better understand what is going on.

5.2 Modes

When reading a joystick input device using C code, there are two modes in which the device can be read: Blocking and Non-Blocking. Each of these modes has a different effect on how the program runs.

For the purposes of explaining these modes, an **event** will refer to any action taken on a joystick input device. Any changes in axis values or button states counts as an event. It is important to note that event reading occurs within the main while loop of each program.

Blocking: This mode will wait for an event to occur before executing any code following the event read. In the case of a while loop, the event read will essentially pause the loop until an event has occurred.

Non-Blocking: This mode will not wait for an event to occur before executing any code following the event read. In the case of a while loop, the loop will continue to run regardless of any events occurring or not occurring.

5.3 Project Programs

There are a total of four sample programs provided in the GitHub repository for this project. There are two **reading** programs and two **testing** programs.

Reading, Blocking and Non-Blocking: The reading programs demonstrate the basics of reading a joystick input device. These programs mimic the code used to test joysticks using the `jstest` command from the Joystick Testing and Configuration Tool. Both programs list all available axes and buttons with their corresponding number and value/state. The user is encouraged to run and explore both programs, since each will behave differently according to their mode.

Testing, Blocking and Non-Blocking: The testing programs use the `rc_project_template.c` file to show the user where the code from the reading programs would be inserted. These programs use BUTTON 0 to control the states of the green and red LEDs on the BeagleBone Blue. They also have a counter that displays a value that the user can change by holding down BUTTON 1 (or AXIS 5) and moving AXIS 1.

Again, the user is encouraged to run and explore both programs. Each mode will make the program behave differently. It is up to the user to decide which mode will work best with their project.

5.4 Code

The following pages contain the code for all four programs provided in the GitHub repository for this project.

read_joy_blocking.c

```
/**
 * @file read_joy_blocking.c
 *
 * Read input from a joystick (/dev/input/js0)
 * Created w/aid of Joystick API Documentation
 * For BeagleBone Blue
 *
 * Manuel Saldana
 *
 */

#include <sys/ioctl.h>
#include <sys/time.h>
#include <sys/types.h>
#include <stdlib.h>
#include <fcntl.h>
#include <unistd.h>
#include <stdio.h>
#include <errno.h>
#include <string.h>
#include <stdlib.h>
#include <stdint.h>
#include <linux/input.h>
#include <linux/joystick.h>

int main()
{
        // INIT
        // open device [in blocking mode]
        // fd is file descriptor
        int fd = open("/dev/input/js0", O_RDONLY);

        // IOCTLs
        // read number of axes
        char number_of_axes;
        ioctl(fd, JSIOCGAXES, &number_of_axes);
        // read number of buttons
        char number_of_buttons;
        ioctl(fd, JSIOCGBUTTONS, &number_of_buttons);

        // OTHER VARIABLES [NOT PART OF API DOC]
        // create pointers for axes and buttons
        int *axis;
        char *button;
        // allocate memory for pointers
        axis = calloc(number_of_axes, sizeof(int));
        button = calloc(number_of_buttons, sizeof(char));

        // WHILE LOOP
        // loop that will continue to run
        while(1)
        {

                // EVENT READING
                // creating a js_event struct called e
                // (e is of type js_event)
                // js_event is defined in joystick.h
```

```
struct js_event e;
// read data previously written to a file
// read sizeof(e) bytes from file descriptor fd into buffer pointed to by &e
read(fd, &e, sizeof(e));

// JS_EVENT.TYPE
// possible values for type are defined in joystick.h
//              button pressed
//              joystick moves
//              init state of device

// JS_EVENT.NUMBER
// values of number correspond to axis or button that generated event
// values vary from one joystick to another

// JS_EVENT.VALUE
// for axis, value is position of joystick along axis
// axis values range from -32767 to 32767
// for buttons, value is state of button
// button values range from 0 to 1

// JS_EVENT.TIME
// values of time correspond to the time an event was generated

// DETERMINE AND ASSIGN [NOT PART OF API DOC]
// determine if event is of type button or axis
// assign value to corresponding button or axis
switch(e.type & ~JS_EVENT_INIT)
{
        case JS_EVENT_BUTTON:
                button[e.number] = e.value;
                break;

        case JS_EVENT_AXIS:
                axis[e.number] = e.value;
                break;
}

// DISPLAY VALUES
// move to beginning of current line
printf("\r");
// display axis numbers and values
if(number_of_axes)
{
        printf("AXES:␣");
        for(int i = 0; i < number_of_axes; i++)
        {
                printf("%2d:%6d␣", i, axis[i]);
        }
}
// display button numbers and values
if(number_of_buttons)
{
        printf("BUTTONS:␣");
        for(int i = 0; i < number_of_buttons; i++)
        {
                printf("%2d:%s␣", i, button[i] ? "1":"0");
        }
}
```

```
                    // flush
                    fflush(stdout);

          }

}
```

read_joy_nonblock.c

```
/**
 * @file read_joy_nonblock.c
 *
 * Read input from a joystick (/dev/input/js0)
 * Created w/aid of Joystick API Documentation
 * For BeagleBone Blue
 *
 * Manuel Saldana
 *
 */

#include <sys/ioctl.h>
#include <sys/time.h>
#include <sys/types.h>
#include <stdlib.h>
#include <fcntl.h>
#include <unistd.h>
#include <stdio.h>
#include <errno.h>
#include <string.h>
#include <stdlib.h>
#include <stdint.h>
#include <linux/input.h>
#include <linux/joystick.h>

int main()
{
        // INIT
        // open device [in non-blocking mode]
        // fd is file descriptor
        int fd = open("/dev/input/js0", O_NONBLOCK);

        // IOCTLs
        // read number of axes
        char number_of_axes;
        ioctl(fd, JSIOCGAXES, &number_of_axes);
        // read number of buttons
        char number_of_buttons;
        ioctl(fd, JSIOCGBUTTONS, &number_of_buttons);

        // OTHER VARIABLES [NOT PART OF API DOC]
        // create pointers for axes and buttons
        int *axis;
        char *button;
        // allocate memory for pointers
        axis = calloc(number_of_axes, sizeof(int));
        button = calloc(number_of_buttons, sizeof(char));

        // WHILE LOOP
        // loop that will continue to run
        while(1)
        {

                // EVENT READING
                // creating a js_event struct called e
                // (e is of type js_event)
                // js_event is defined in joystick.h
```

```
struct js_event e;
// read data previously written to a file
// read sizeof(e) bytes from file descriptor fd into buffer pointed to by &e
// if read returns -1, then there are no events pending to be read
while(read(fd, &e, sizeof(e)) > 0)
{
        // process event
}
// EAGAIN is returned when que is empty
if(errno != EAGAIN)
{
        // error
}

// JS_EVENT.TYPE
// possible values for type are defined in joystick.h
//              button pressed
//              joystick moves
//              init state of device

// JS_EVENT.NUMBER
// values of number correspond to axis or button that generated event
// values vary from one joystick to another

// JS_EVENT.VALUE
// for axis, value is position of joystick along axis
// axis values range from -32767 to 32767
// for buttons, value is state of button
// button values range from 0 to 1

// JS_EVENT.TIME
// values of time correspond to the time an event was generated

// DETERMINE AND ASSIGN [NOT PART OF API DOC]
// determine if event is of type button or axis
// assign value to corresponding button or axis
switch(e.type & ~JS_EVENT_INIT)
{
        case JS_EVENT_BUTTON:
                button[e.number] = e.value;
                break;

        case JS_EVENT_AXIS:
                axis[e.number] = e.value;
                break;
}

// DISPLAY VALUES
// move to beginning of current line
printf("\r");
// display axis numbers and values
if(number_of_axes)
{
        printf("AXES:␣");
        for(int i = 0; i < number_of_axes; i++)
        {
                printf("%2d:%6d␣", i, axis[i]);
        }
}
```

```
                // display button numbers and values
                if(number_of_buttons)
                {
                        printf("BUTTONS:␣");
                        for(int i = 0; i < number_of_buttons; i++)
                        {
                                printf("%2d:%s␣", i, button[i] ? "1":"0");
                        }
                }
                // flush
                fflush(stdout);

        }

}
```

test_joy_blocking.c

```
/**
 * @file test_joy_blocking.c
 *
 * Created using the rc_project_template.c file
 * Read input from a joystick (/dev/input/js0)
 * Control LEDs on BeagleBone Blue
 * Blocking mode
 *
 * Manuel Saldana
 *
 */

#include <stdio.h>

#include <sys/ioctl.h>
#include <sys/time.h>
#include <sys/types.h>
#include <stdlib.h>
#include <fcntl.h>
#include <unistd.h>
#include <errno.h>
#include <string.h>
#include <stdlib.h>
#include <stdint.h>
#include <linux/input.h>
#include <linux/joystick.h>

#include <robotcontrol.h> // includes ALL Robot Control subsystems

// VARIABLES
static int counter;

/**
 * This template contains these critical components
 * - ensure no existing instances are running and make new PID file
 * - start the signal handler
 * - initialize subsystems you wish to use
 * - while loop that checks for EXITING condition
 * - cleanup subsystems at the end
 *
 * @return      0 during normal operation, -1 on error
 */
int main()
{
        // make sure another instance isn't running
        // if return value is -3 then a background process is running with
        // higher privaledges and we couldn't kill it, in which case we should
        // not continue or there may be hardware conflicts. If it returned -4
        // then there was an invalid argument that needs to be fixed.
        if(rc_kill_existing_process(2.0)<-2) return -1;

        // start signal handler so we can exit cleanly
        if(rc_enable_signal_handler()==-1){
                fprintf(stderr,"ERROR:␣failed␣to␣start␣signal␣handler\n");
                return -1;
        }
```

```
// make PID file to indicate your project is running
// due to the check made on the call to rc_kill_existing_process() above
// we can be fairly confident there is no PID file already and we can
// make our own safely.
rc_make_pid_file();

// JOYSTICK INIT
// open device
int fd = open("/dev/input/js0", O_RDONLY);
// read number of axes
char number_of_axes;
ioctl(fd, JSIOCGAXES, &number_of_axes);
// read number of buttons
char number_of_buttons;
ioctl(fd, JSIOCGBUTTONS, &number_of_buttons);
// create pointers for axes and buttons
int *axis;
char *button;
// allocate memory for pointers
axis = calloc(number_of_axes, sizeof(int));
button = calloc(number_of_buttons, sizeof(char));


printf("\nPress␣and␣release␣BUTTON␣0␣to␣turn␣green␣LED␣on␣and␣off\n");
printf("Hold␣BUTTON␣1␣(or␣hold␣AXIS␣5)␣and␣move␣AXIS␣1␣to␣change␣counter␣value\n");
printf("Press␣BUTTON␣3␣to␣exit\n");

// Keep looping until state changes to EXITING
rc_set_state(RUNNING);
while(rc_get_state()!=EXITING){

        // READ FROM JOYSTICK
        struct js_event e;
        read(fd, &e, sizeof(e));
        // determine if event is of type button or axis
        // assign value to corresponding button or axis
        switch(e.type & ~JS_EVENT_INIT)
        {
                case JS_EVENT_BUTTON:
                        button[e.number] = e.value;
                        break;

                case JS_EVENT_AXIS:
                        axis[e.number] = e.value;
                        break;
        }

        // DO THINGS

        // change state of green LED
        if(button[0])
        {
                rc_led_set(RC_LED_GREEN, 1);
                rc_led_set(RC_LED_RED, 0);
        }
        else
        {
                rc_led_set(RC_LED_GREEN, 0);
                rc_led_set(RC_LED_RED, 1);
                }

                // change value of counter
                if(button[1] || axis[5] > -32767)
                {
                        if(axis[1] > 0)
                        {
                                counter++;
                        }
                        else if(axis[1] < 0)
                        {
                                counter--;
                        }
                }

                // exit program
                if(button[3])
                {
                        rc_set_state(EXITING);
                        printf("\nGoodbye...\n");
                        continue;
                }

                // display values of specific buttons and axes
                printf("\r");
                printf("BUTTONS:␣");
                printf("%2d:%s␣", 0, button[0] ? "1":"0");
                printf("%2d:%s␣", 1, button[1] ? "1":"0");
                printf("%2d:%s␣", 3, button[3] ? "1":"0");
                printf("␣|␣AXES:␣");
                printf("%2d:%6d␣", 1, axis[1]);
                printf("%2d:%6d␣", 5, axis[5]);
                printf("␣|␣COUNTER:␣");
                printf("%d␣", counter);
                fflush(stdout);

                // always sleep at some point
                rc_usleep(100000);
        }

        // turn off LEDs and close file descriptors
        rc_led_set(RC_LED_GREEN, 0);
        rc_led_set(RC_LED_RED, 0);
        rc_led_cleanup();
        rc_button_cleanup();    // stop button handlers
        rc_remove_pid_file();   // remove pid file LAST
        return 0;
}
```

test_joy_nonblock.c

```
/**
 * @file test_joy_nonblock.c
 *
 * Created using the rc_project_template.c file
 * Read input from a joystick (/dev/input/js0)
 * Control LEDs on BeagleBone Blue
 * Non-blocking mode
 *
 * Manuel Saldana
 *
 */

#include <stdio.h>

#include <sys/ioctl.h>
#include <sys/time.h>
#include <sys/types.h>
#include <stdlib.h>
#include <fcntl.h>
#include <unistd.h>
#include <errno.h>
#include <string.h>
#include <stdlib.h>
#include <stdint.h>
#include <linux/input.h>
#include <linux/joystick.h>

#include <robotcontrol.h> // includes ALL Robot Control subsystems

// VARIABLES
static int counter;

/**
 * This template contains these critical components
 * - ensure no existing instances are running and make new PID file
 * - start the signal handler
 * - initialize subsystems you wish to use
 * - while loop that checks for EXITING condition
 * - cleanup subsystems at the end
 *
 * @return      0 during normal operation, -1 on error
 */
int main()
{
        // make sure another instance isn't running
        // if return value is -3 then a background process is running with
        // higher privaledges and we couldn't kill it, in which case we should
        // not continue or there may be hardware conflicts. If it returned -4
        // then there was an invalid argument that needs to be fixed.
        if(rc_kill_existing_process(2.0)<-2) return -1;

        // start signal handler so we can exit cleanly
        if(rc_enable_signal_handler()==-1){
                fprintf(stderr,"ERROR:␣failed␣to␣start␣signal␣handler\n");
                return -1;
        }
```

```
// make PID file to indicate your project is running
// due to the check made on the call to rc_kill_existing_process() above
// we can be fairly confident there is no PID file already and we can
// make our own safely.
rc_make_pid_file();

// JOYSTICK INIT
// open device
int fd = open("/dev/input/js0", O_NONBLOCK);
// read number of axes
char number_of_axes;
ioctl(fd, JSIOCGAXES, &number_of_axes);
// read number of buttons
char number_of_buttons;
ioctl(fd, JSIOCGBUTTONS, &number_of_buttons);
// create pointers for axes and buttons
int *axis;
char *button;
// allocate memory for pointers
axis = calloc(number_of_axes, sizeof(int));
button = calloc(number_of_buttons, sizeof(char));


printf("\nPress␣and␣release␣BUTTON␣0␣to␣turn␣green␣LED␣on␣and␣off\n");
printf("Hold␣BUTTON␣1␣(or␣hold␣AXIS␣5)␣and␣move␣AXIS␣1␣to␣change␣counter␣value\n");
printf("Press␣BUTTON␣3␣to␣exit\n");

// Keep looping until state changes to EXITING
rc_set_state(RUNNING);
while(rc_get_state()!=EXITING){

        // READ FROM JOYSTICK
        struct js_event e;
        while(read(fd, &e, sizeof(e)) > 0)
        {
                // process event
        }
        // EAGAIN is returned when que is empty
        if(errno != EAGAIN)
        {
                // error
        }
        // determine if event is of type button or axis
        // assign value to corresponding button or axis
        switch(e.type & ~JS_EVENT_INIT)
        {
                case JS_EVENT_BUTTON:
                        button[e.number] = e.value;
                        break;

                case JS_EVENT_AXIS:
                        axis[e.number] = e.value;
                        break;
        }

        // DO THINGS

        // change state of green LED
        if(button[0])
```

```
                {
                        rc_led_set(RC_LED_GREEN, 1);
                        rc_led_set(RC_LED_RED, 0);
                }
                else
                {
                        rc_led_set(RC_LED_GREEN, 0);
                        rc_led_set(RC_LED_RED, 1);
                }

                // change value of counter
                if(button[1] || axis[5] > -32767)
                {
                        if(axis[1] > 0)
                        {
                                counter++;
                        }
                        else if(axis[1] < 0)
                        {
                                counter--;
                        }
                }

                // exit program
                if(button[3])
                {
                        rc_set_state(EXITING);
                        printf("\nGoodbye...\n");
                        continue;
                }

                // display values of specific buttons and axes
                printf("\r");
                printf("BUTTONS:␣");
                printf("%2d:%s␣", 0, button[0] ? "1":"0");
                printf("%2d:%s␣", 1, button[1] ? "1":"0");
                printf("%2d:%s␣", 3, button[3] ? "1":"0");
                printf("␣|␣AXES:␣");
                printf("%2d:%6d␣", 1, axis[1]);
                printf("%2d:%6d␣", 5, axis[5]);
                printf("␣|␣COUNTER:␣");
                printf("%d␣", counter);
                fflush(stdout);

                // always sleep at some point
                rc_usleep(100000);
        }

        // turn off LEDs and close file descriptors
        rc_led_set(RC_LED_GREEN, 0);
        rc_led_set(RC_LED_RED, 0);
        rc_led_cleanup();
        rc_button_cleanup();    // stop button handlers
        rc_remove_pid_file();   // remove pid file LAST
        return 0;
}
```

L2: Axis 2 [-32767, 32767]
L2: Button 6 [on, off]

L1: Button 4 [on, off]

SHARE: Button 8 [on, off]

OPTIONS: Button 9 [on, off]

R2: Axis 5 [-32767, 32767]
R2: Button 7 [on, off]

R1: Button 5 [on, off]

UP: Axis 7 [-32767, 0]

DOWN: Axis 7 [0, 32767]

LEFT: Axis 6 [-32767, 0]

RIGHT: Axis 6 [0, 32767]

X: Button 0 [on, off]

CIRCLE: Button 1 [on, off]

TRIANGLE: Button 2 [on,off]

SQUARE: Button 3 [on, off]

L,X: Axis 0 [-32767, 32767]

L,Y: Axis 1 [-32767, 32767]

PS: Button 10 [on, off]

L3: Button 11 [on, off]

R3: Button 12 [on, off]

R,X: Axis 3 [-32767, 32767]

R,Y: Axis 4 [-32767, 32767]

LT: Axis 5 [-32767, 32767]

LB: Button 6 [on, off]

VIEW: NONE

MENU: Button 11 [on, off]

RT: Axis 4 [-32767, 32767]

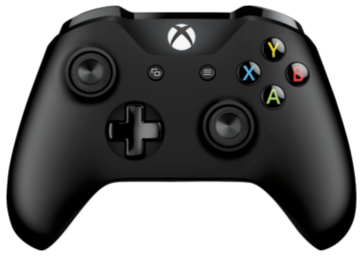
RB: Button 7 [on, off]

UP: Axis 7 [-32767, 0]

DOWN: Axis 7 [0, 32767]

LEFT: Axis 6 [-32767, 0]

RIGHT: Axis 6 [0, 32767]



A: Button 0 [on, off]

B: Button 1 [on, off]

X: Button 3 [on, off]

Y: Button 4 [on, off]

L,X: Axis 0 [-32767, 32767]

L,Y: Axis 1 [-32767, 32767]

XBOX: NONE

L STICK: Button 13 [on, off]

R STICK: Button 14 [on, off]

R,X: Axis 2 [-32767, 32767]

R,Y: Axis 3 [-32767, 32767]

Figure 3: DualShock 4 and Xbox One S Controller Mappings