This document shall outline the design of my DIY electric skateboard. There are two sections in this report, high-level system design and firmware design and documentation.

Section 1) High Level-System Design:

The figure below shows a high-level diagram of the electric longboard hardware. The battery pack supply power to the entire system from the motor drive power circuit to low voltage electronics. The battery pack is directly connected to the motor drive power circuit which has a built-in 5 Volts regulator. This 5 Volts source is used to power the microcontroller unit (MCU). Similarly, the MCU has 3.3 V regulator that powers its circuits and the RC receiver circuit.

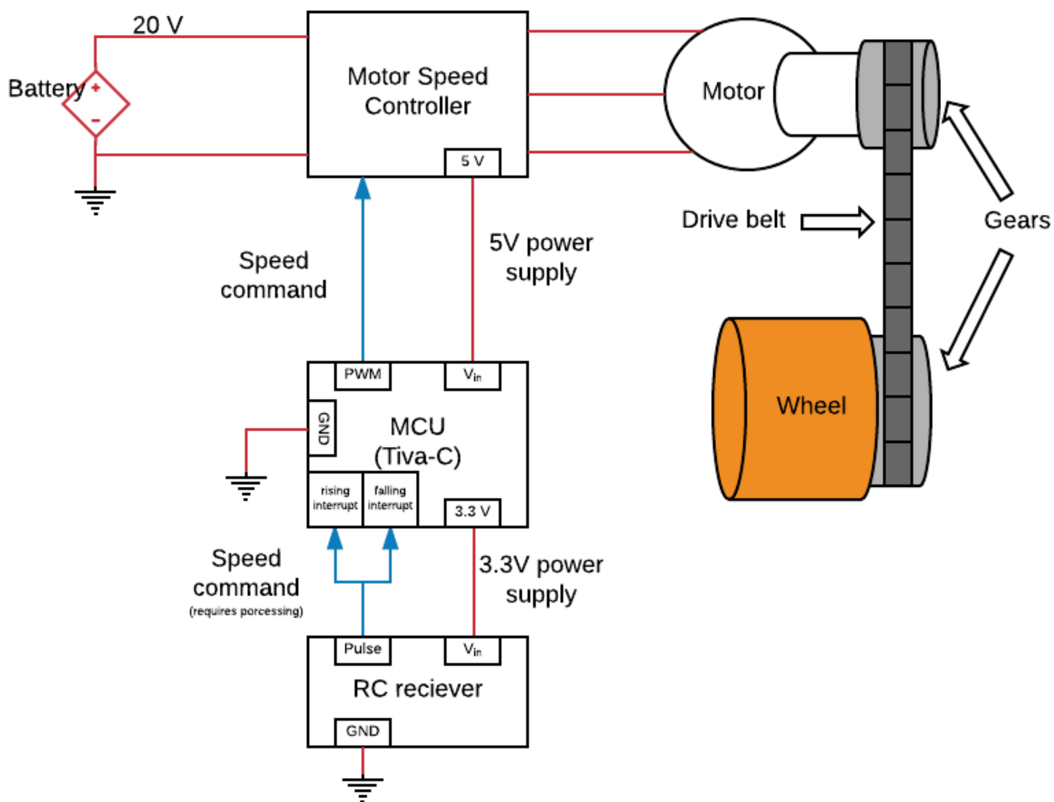

Figure 1: High-level system diagram of E-longboard electro-mechanical system

It was not possible to connect the RC receiver directly to the motor drive power circuit, as the RC speed command signal does meet the specifications of the required motor drive speed command signal. The motor drive command signal expects a PWM signal with a duty cycle between 50% and 100%. The 50% and 100% duty cycles correspond to zero speed and full speed commands, respectively. However, the RC receiver generates a PWM signal with a duty cycle range of about 15 to 30%. Consequently, the MCU is used to bridge to RC signal to the motor drive circuit and perform all the necessary duty cycle value conversion.

The RC receiver speed command signal is connected to two GPIO pins on the MCU. These pins are used to asset interrupts that enable a timer on MCU to measure the width of the pulse generated by the RC receiver. Consequently, the MCU translates the speed command of the RC receiver into a PWM signal that will range from 50% to 100% to command the motor drive speed reference of the motor drive.

In regards to the mechanical drive system, the motor drives one of the rear wheels of the longboard. The motor is coupled to the wheel through a pulley-belt system. The larger pulley is attached to the wheel to generate higher torque at the wheel. Figure shows a picture of the pulley-belt system.

I'll add the picture of the longboard, as so as I weld the motor mount to the longboard trucks.

Section 2) Firmware:

The figure below shows a hierarchical diagram of how the firmware source code modules interreact. Peripheral drivers provide APIs for application source code to use. Description of the drivers and applications API's is provided in the appendix. However, Table 1 provides a brief description of the firmware source code file groups.

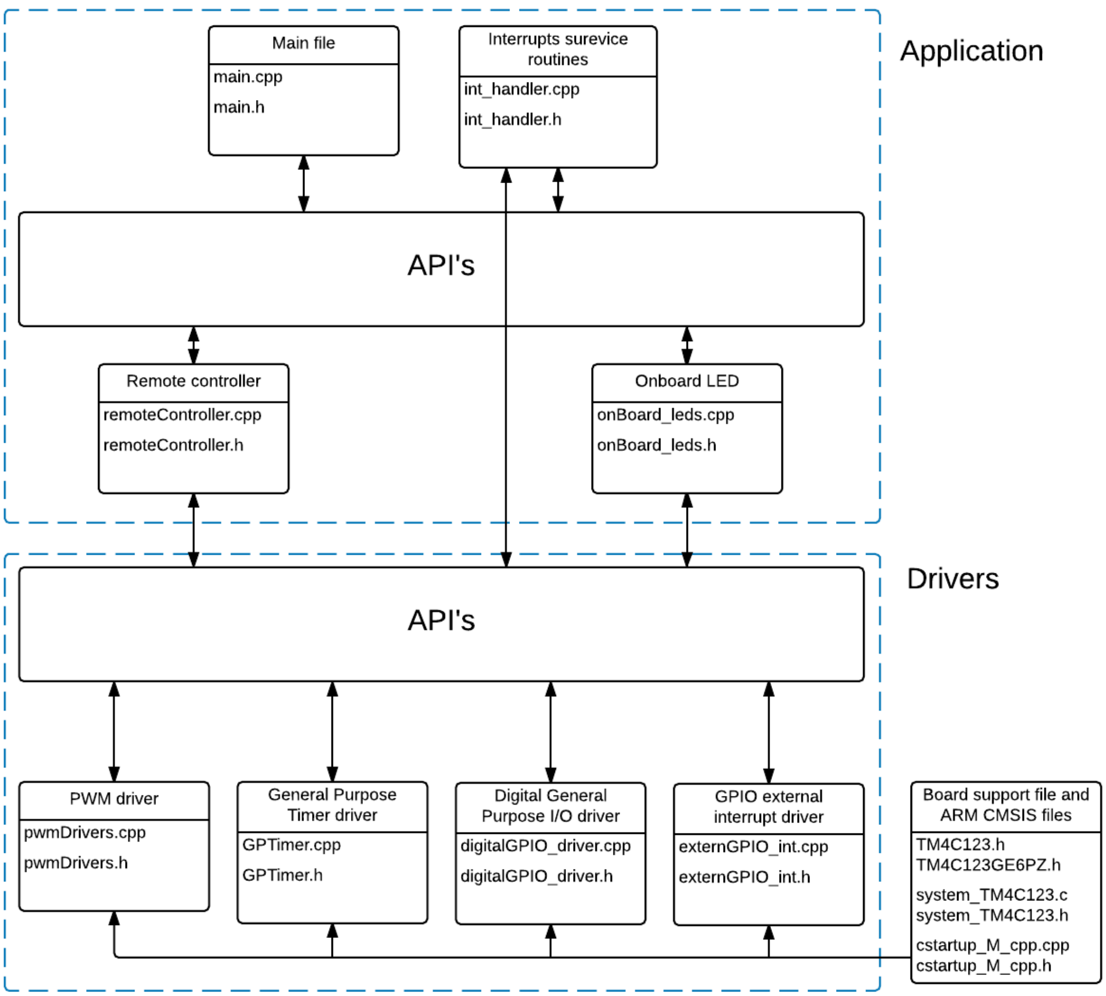

Figure 2: High-level diagram of firmware

Table 1: summary of source code files

Files group	Description
Main file	Performs system and drivers initializations by invoking API provided by application and drivers source code, and then enters an infinite empty loop.
Interrupt service routines	Define all system interrupt service routines: • SysTick interrupt, which asserts an interrupt every 10 msec to performed tasks. • GPIO external edge detection interrupt; this interrupt is used to enable a timer to measure the width of RC receiver generated signal.
Remote controller	Define application functions to do the following: • Configure any pair GPIO pin from the same port as external edge detection interrupt. • Enable or disable the GPIO pin interrupt by masking them. • Mapping function that linear map a number from one range to another; this function is used to transform the timer value that measures the width of the RC controller pulse signal to a duty cycle of PWM signal command to the motor drive.
Onboard LED	Define application function to toggle and blink on and off the LEDs on the Tiva-C development board. It's used for debugging purposes only.
PWM driver	Provide driver configuration and initialization, as well as the following APIs: • Enabling and disabling PWM pins. • Changing the PWM frequency and duty cycle.
General purpose timer driver	Provide driver configuration and initialization for the 32 bit timer0 on Tiva-C board, as well as the following APIs: • Enabling and disabling the timer counter. • Manually setting the value of the timer counter. • Reading the value the timer counter.
Digital General purpose I/O driver	Provide driver configuration and initialization, as well as the following APIs: • Setting the value of a GPIO pin. • Reading the value of a GPIO pin.
GPIO external interrupt driver	Provide driver configuration and initialization, as well as the following APIs: • Masking interrupts. • Enabling and disabling interrupts. • Setting the priority of the interrupt.

The following is a description of the execution flow of the firmware. The main.cpp file performs a one-time sequence of initializations and configurations for system peripherals and then enters an infinite empty loop. Further, a system time (SysTick) interrupt is asserted every 10 msec. The SysTick interrupt poll over four windows with 10 msec timer period; thus, effectively, tasks in these interrupt windows get executed every 40 msec. Figure 3 shows a timing diagram of the SysTick interrupt.

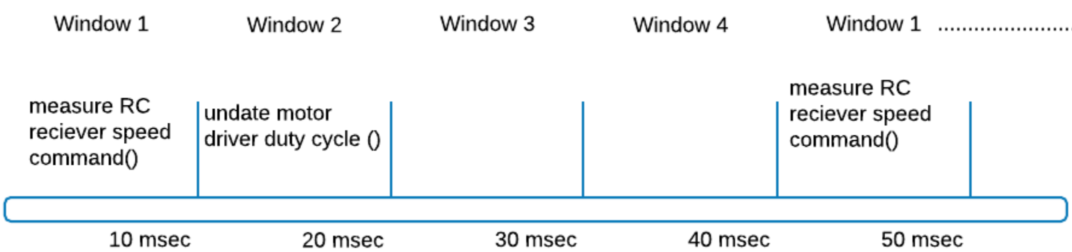

Figure 3: SysTick interrupt timing diagram

Currently, only two tasks are executed in the SysTick interrupt. The first task is to measure the speed command supplied by the RC receiver. This task is accomplished by unmasking the external edge detection interrupts that control the timer that measures the width of the received signal from RC receiver. This first task is performed during the first window of the SysTick interrupt.

The second task updates the duty cycle value of the PWM signal that is fed to the motor drive power circuit. This task is performed in the second window of the SysTic interrupt

Appendix

This appendix provides a detailed description of the firmware drivers' APIs.

PWM driver:

This driver configures and initializes the PWM module. The driver provides the following API to application layer:

1) PWM module configuration and initialization.
2) Enabling and disabling the PWM module.
3) Specifying the frequency and duty cycle of the PWM.

The following is a description of the functions in the PWM API:

Data structure:

This data structure allows the application layer to specify the GPIO pin, frequency, and duty cycle of the PWM output.

```
typedef struct {
	uint32_t pin;    // GPIO pin that generate the PWM signal
	uint32_t freq;   // frequency of the PWM signal
	float duty;      // duty cycle of the PWM signal; range from 0 to 100

} pwm_pin_config_t;
```

Initialization and configuration:

This function allows the application layer to initialize the PWM module.

```
/*****************************************************************
* @brief      This function initializes the PWM pin and module
* @param      *pwm: this is a pointer to the PWM module
* @param      *GPIOx: pointer the GPIO port
* @param      configStruct: structure that has pin configuration info
* @return None
*****************************************************************/
void dr_pwm_init(PWM0_Type *pwmx, GPIOA_Type *GPIOx, pwm_pin_config_t *configStruct);
```

The following is an example of how to initialize *PWM0* module in the Texas Instrument-based ARM cortex-M4. Pin 6 on GPIO port B was initialized with 5 kHz frequency and 50% duty cycle.

```
/*------Configuring PWM ------*/
pwm_pin_config_t pwmSpecs;
pwmSpecs.pin = 6;
pwmSpecs.freq = 5000;
pwmSpecs.duty = 50;

dr_pwm_init(PWM0, GPIOB_AHB, &pwmSpecs);
```

Enabling PWM signal on a particular pin:

```
/***************************************************************
* @brief      Enables pwm pin
* @param      *pwm: this is a pointer to the PWM module
* @param      *GPIOx: pointer the GPIO port
* @param      pin: pwm pin to be enabled
* @return None
***************************************************************/
void dr_pwm_enable(PWM0_Type *pwmx, GPIOA_Type *GPIOx, uint32_t pin);
```

Disabling PWM signal on a particular pin:

```
/***************************************************************
* @brief      Disbales pwm pin
* @param      *pwm: this is a pointer to the PWM module
* @param      *GPIOx: pointer the GPIO port
* @param      pin: pwm pin to be disabled
* @return None
***************************************************************/
void dr_pwm_disable(PWM0_Type *pwmx, GPIOA_Type *GPIOx, uint32_t pin);
```

Changing the value and frequency of the PWM during run time:

Unlike the initialization function, this function does have to momentarily disable the PWM pin.

```
/***************************************************************
* @brief      Set the value of PWM
* @param      *pwm: this is a pointer to the PWM module
* @param      *GPIOx: pointer the GPIO port
* @param      configStruct: structure that has pin configuration info
* @return None
***************************************************************/
void dr_pwm_freq_duty_set(PWM0_Type *pwmx, GPIOA_Type *GPIOx, pwm_pin_config_t *configStruct);
```

General Purpose timer driver:

This driver configures and initializes the general-purpose timer module. The driver provides the following APIs:

1) General purpose timer module configuration and initialization
2) Enabling and disabling the timer
3) Manually setting the value of the timer counter
4) Reading the value of the timer counter

Macros used by API:

```
// enalble to disable timer0
#define TIMER0_DISABLE      (0U)
#define TIMER0_ENABLE       (1U)
```

Initialization and configuration:

```
/*******************************************************************
* @brief      Initilize timer0 as a 32 bit timer
* @param      None
* @return     None
*******************************************************************/
void dr_timer0_init_32();
```

Enable or Disable the timer:

```
/*******************************************************************
* @brief      Enable or Disable the timer.
* @param      enb: this take one of the defined directives above
* @return     None
*******************************************************************/
void dr_timer0_enable(uint32_t enb);
```

Manually set the value of the timer's counter:

```
/*******************************************************************
* @brief      Manually set the value of timers
* @param      val: value the timer should be set to
* @return     None
*******************************************************************/
void dr_timer0_set_val(uint32_t val);
```

Read the value of the timer's counter:

```
/*****************************************************************
* @brief  Read the value of timer
* @param      none
* @return the value of the timer
*****************************************************************/
uint32_t dr_timer0_read_val();
```

Digital GPIO driver:

This driver configures and initializes the digital functionality of GPIO pins. The driver provides the following APIs:

1) GPIO pin configuration and initialization
2) Reading the value of a GPIO pin
3) Setting the value of a GPIO pin

Macros:

```
// GPIO degital
#define DIGITAL_DISABLE (0U)
#define DIGITAL_ENABLE  (1U)

// GPIO direction
#define GPIO_DIR_INPUT  (0U)
#define GPIO_DIR_OUTPUT (1U)

// GPIO pull-up resistor
#define GPIO_PULLUP_RESISTOR_FALSE  (0U)
#define GPIO_PULLUP_RESISTOR_TRUE   (1U)

// GPIO pull-down resistor
#define GPIO_PULLDOWN_RESISTOR_FALSE  (0U)
#define GPIO_PULLDOWN_RESISTOR_TRUE   (1U)

// GPIO open drain
#define GPIO_OPEN_DRAIN_FALSE (0U)
#define GPIO_OPEN_DRAIN_TRUE  (1U)

// function Macros to enable clocks to GPIO ports
#define _GPIOA_CLK_ENABLE()     (SYSCTL->RCGC2 |= (1U<<0) )
#define _GPIOB_CLK_ENABLE()     (SYSCTL->RCGC2 |= (1U<<1) )
#define _GPIOC_CLK_ENABLE()     (SYSCTL->RCGC2 |= (1U<<2) )
#define _GPIOD_CLK_ENABLE()     (SYSCTL->RCGC2 |= (1U<<3) )
#define _GPIOE_CLK_ENABLE()     (SYSCTL->RCGC2 |= (1U<<4) )
#define _GPIOF_CLK_ENABLE()     (SYSCTL->RCGC2 |= (1U<<5) )

// function Macros to enable AHB
#define _GPIOA_AHB_ENABLE()     (SYSCTL->GPIOHBCTL |= (1U<<0) )
#define _GPIOB_AHB_ENABLE()     (SYSCTL->GPIOHBCTL |= (1U<<1) )
#define _GPIOC_AHB_ENABLE()     (SYSCTL->GPIOHBCTL |= (1U<<2) )
#define _GPIOD_AHB_ENABLE()     (SYSCTL->GPIOHBCTL |= (1U<<3) )
#define _GPIOE_AHB_ENABLE()     (SYSCTL->GPIOHBCTL |= (1U<<4) )
#define _GPIOF_AHB_ENABLE()     (SYSCTL->GPIOHBCTL |= (1U<<5) )
```

Data structure:

```c
// Application should use Macros defined above to specify pin driver and configuration

typedef struct {

	uint32_t pin;                 // Specify the GPIO pin to be configured
	uint32_t direction;           // Specify input or output
	uint32_t pull_up_resis;       // Specify to add internal pull-up resistor
	uint32_t pull_down_resis;     // Specify to add internal pull-down resistor
	uint32_t open_drain;          // Specify to configure as open drain or not

} gpio_digital_pin_conf_t;
```

Initialization and configuration:

```c
/*****************************************************************
* @brief      Initializes the GPIO pin
* @param      *GPIOx: base address of GPIO Port
* @param      *gpio_digital_pin_conf_t: pointer to the pin conf structure sent by
application
* @return     None
*****************************************************************/
void dr_gpio_digital_init(GPIOA_Type *GPIOx, gpio_digital_pin_conf_t *gpio_pin_conf)
```

The following is an example of how to initialize pin 1 of GPIO port F as an output pin without a pullup/down internal resistors and with no open mode drain.

```c
_GPIOF_CLK_ENABLE(); // enable clock to GPIO port F
_GPIOF_AHB_ENABLE(); // enable clock to AHB

// fill in the structure to configure the GPIO port
gpio_digital_pin_conf_t gpio_pin_conf;
gpio_pin_conf.direction = GPIO_DIR_OUTPUT;
gpio_pin_conf.pull_down_resis = GPIO_PULLDOWN_RESISTOR_FALSE;
gpio_pin_conf.pull_up_resis = GPIO_PULLUP_RESISTOR_FALSE;
gpio_pin_conf.open_drain = GPIO_OPEN_DRAIN_FALSE;
gpio_pin_conf.pin = 1;

// initializing red led pin in GPIO port F
dr_gpio_digital_init(GPIOF_AHB, &gpio_pin_conf);
```

Read bit value of a GPIO port:

```c
/*****************************************************************
* @brief      Read the value of the pin
* @param      *GPIOx: base address of GPIO Port
* @param      pin: pin number to be read
* @return     uint32_t with a value of 0 or 1
*****************************************************************/
uint32_t dr_pin_digital_read(GPIOA_Type *GPIOx, uint32_t pin);
```

Write to a bit in a GPIO port:

```
/*****************************************************************
* @brief      Write a value to the pin
* @param      *GPIOx: base address of GPIO Port
* @param      pin: pin number to be written to
* @param      val: value to be written to pin; 0 or 1
* @return     None
*****************************************************************/
void dr_pin_digital_write(GPIOA_Type *GPIOx, uint32_t pin, uint32_t val);
```

GPIO external interrupt driver:

This driver is built on top of the digital GPIO driver. This driver configures a GPIO pin for external edge or voltage level interrupt detection. It provides the following APIs:

1) Interrupt hardware initialization
2) Masking and unmasking interrupts
3) Clearing interrupt flags
4) Enabling and disabling interrupts
5) Setting the priorities of interrupts

Macros:

```
/*********************************************************************************
*
*            1. Macros used for GPIO pin Initialization
*
*********************************************************************************/
// edge or voltage level detection
#define EDGE_INT   (0U)
#define LEVEL_INT  (1U)

// detect on which edge
#define FALLING_EDGE  (0U)
#define RISING_EDGE   (1U)
#define RISING_AND_FALLING_EDGE  (2U)

// Masking interupprts
#define MASK_INT_TRUE     (0U)
#define MASK_INT_FALSE    (1U)
```

Data Structure:

```
// Application should use Macros defined above to specify pin driver and configuration

typedef struct {

       uint32_t pin;           // Specify the GPIO pin to be configured
       uint32_t edgeOrLevel;   // Specify whether to detect on edge or voltage level
       uint32_t edgeTriger;    // Specify on which edge to trigger the interrupt on
       uint32_t priority;      // Specify the priority of the interrupt 0-7

} gpio_extern_int_conf_t;
```

Initialization and configuration:

```
/****************************************************************
* @brief  Initializes the GPIO pin to be configured as an interrupt
* @param  *GPIOx: base address of GPIO Port
* @param  *gpio_pin_conf: pointer to the pin conf structure sent by application
* @return None
****************************************************************/
void dr_gpio_extern_int_init(GPIOA_Type *GPIOx, gpio_extern_int_conf_t *gpio_pin_conf);
```

The following is an example of how to initialize pin 0 of GPIO port F as a falling edge detection external interrupt.

```
_GPIOF_CLK_ENABLE(); // enable clock to GPIO port F
_GPIOF_AHB_ENABLE(); // enable clock to AHB


gpio_extern_int_conf_t externalInt;
externalInt.pin = 0;
externalInt.edgeOrLevel = EDGE_INT;       // edge detection interrupt
externalInt.edgeTriger = FALLING_EDGE;    // falling edge detection
externalInt.priority = 2;                 // setting propriety
dr_gpio_extern_int_init(GPIOF_AHB, &externalInt);
```

Masking interrupts:

```
/****************************************************************
  * @brief  This API allows the user to mask/unmask GPIO interrupts
  * @param  *GPIOx: base address of GPIO Port
  * @param  pin: pin number to be read
  * @param  mask: specify whether on not mask an interrupt; use macros define above
  * @return None
****************************************************************/
void dr_gpio_extern_int_mask(GPIOA_Type *GPIOx, uint32_t pin, uint32_t mask);
```

Clearing interrupt flag to enable new interrupt assertion:

```
/****************************************************************
  * @brief  Clear the interrupt status register to allow interrupts to be triggered again
  * @param  *GPIOx: base address of GPIO Port
  * @param  pin: pin number to be written to
  * @return None
****************************************************************/
void dr_gpio_extern_int_clear(GPIOA_Type *GPIOx, uint32_t pin);
```

Enabling interrupt on a particular GPIO port:

```
/**************************************************************
  * @brief  Enable interrupt to GPIO port
  * @param  *GPIOx: pointer the GPIO port
  * @return None
**************************************************************/
void dr_gpio_extern_int_enable(GPIOA_Type *GPIOx);
```

Disabling interrupt on a particular GPIO port:

```
/**************************************************************
  * @brief  Disable interrupt to GPIO port
  * @param  *GPIOx: pointer the GPIO port
  * @return None
**************************************************************/
void dr_gpio_extern_int_disable(GPIOA_Type *GPIOx);
```

Setting the priority of an interrupt:

```
/**************************************************************
  * @brief  Set priotery level for GPIO port
  * @param  *GPIOx: base address of GPIO Port
  * @param  priotery: set priotery level; takes value from 0-7
  * @return None
**************************************************************/
void dr_gpio_extern_int_priotery(GPIOA_Type *GPIOx, uint32_t priotery);
```