Parker Brown
A11961591
MAE 144

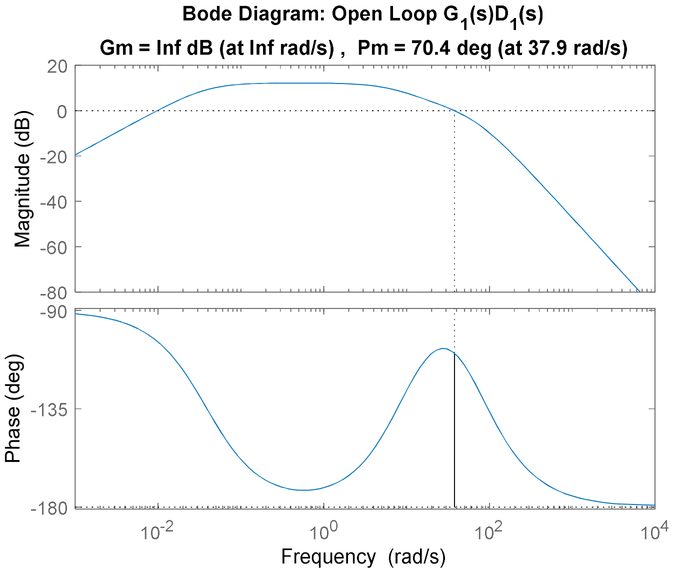
Final Project: Balance

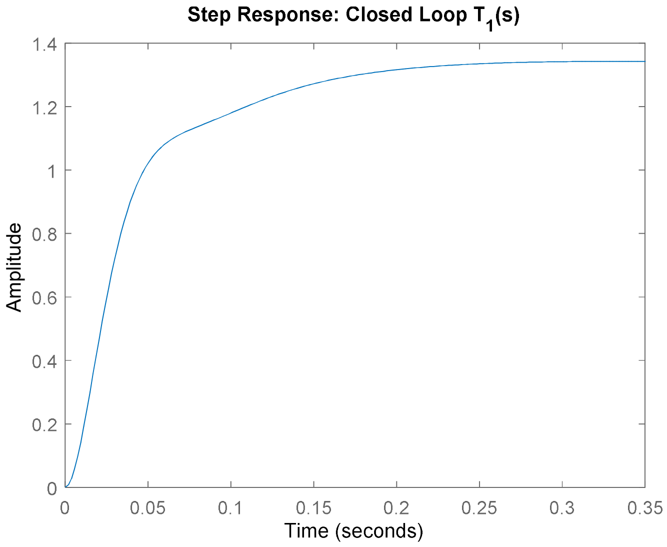
<u>G1:</u> Inner Loop

$$G_1(s) = -\frac{875.6s}{s^3 + 44.18s^2 - 143.8s - 2072}$$

$$D_1(s) = \frac{-4.95s^2 - 141.1s - 705.9}{s^2 + 73.15s + 2.822}$$

$$D_1(z) = \frac{-4.9500z^2 + 8.8709z - 3.9709}{z^2 - 1.4810z + 0.4812} \; at\; 100Hz$$


Bode Diagram: Open Loop $G_1(s)D_1(s)$
Gm = Inf dB (at Inf rad/s) , Pm = 70.4 deg (at 37.9 rad/s)

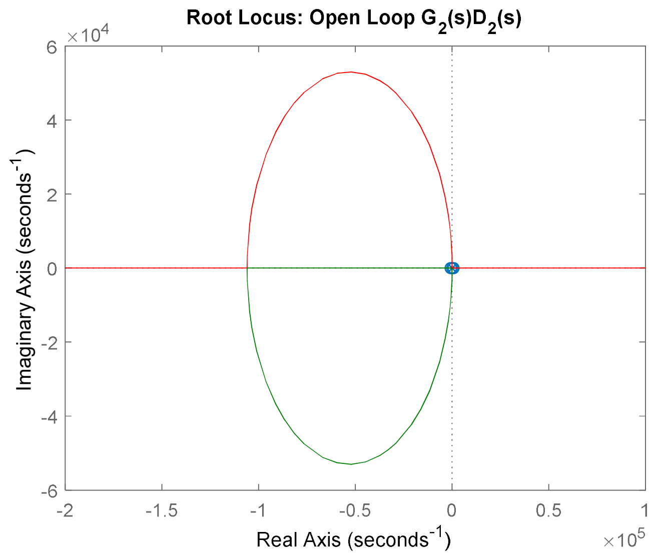

Step Response: Closed Loop $T_1(s)$

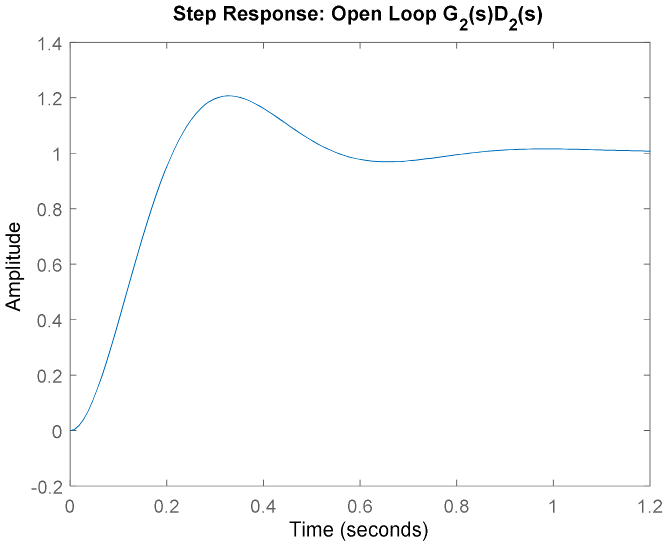
G2: Outer Loop

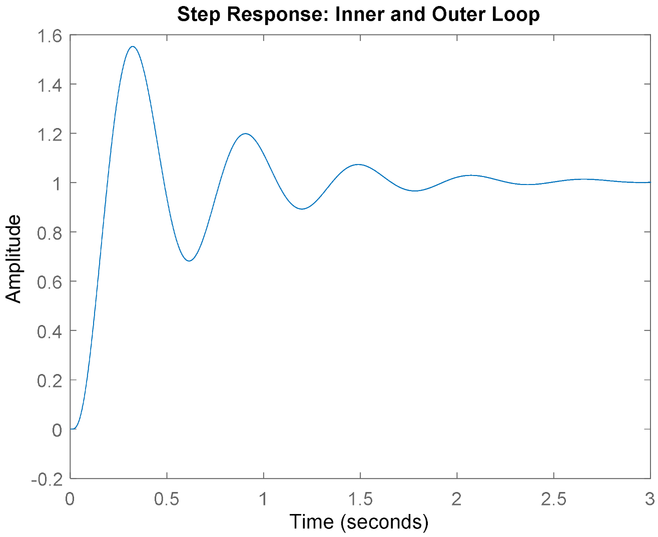
$$G_2(s) = \frac{-0.0002233s^2 + 117.1}{s^2}$$

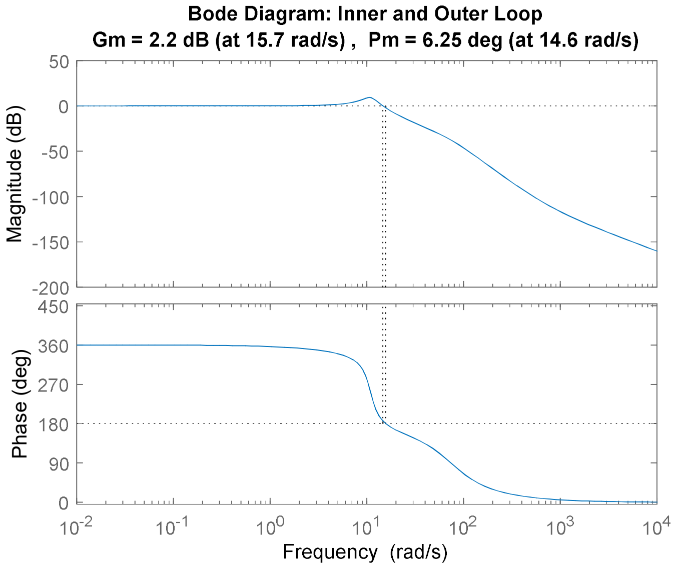
$$D_2(s) = \frac{s + 0.1}{s + 10}$$

$$D_2(z) = \frac{z - 0.9961}{z - 0.6065} \; at \; 20Hz$$






Step Response: Inner and Outer Loop
Amplitude
1.6
1.4
1.2
1
0.8
0.6
0.4
0.2
0
-0.2
0
0.5
1
1.5
2
2.5
3
Time (seconds)


Bode Diagram: Inner and Outer Loop
Gm = 2.2 dB (at 15.7 rad/s) , Pm = 6.25 deg (at 14.6 rad/s)
Magnitude (dB)
50
0
-50
-100
-150
-200
Phase (deg)
450
360
270
180
90
0
10^{-2}
10^{-1}
10^{0}
10^{1}
10^{2}
10^{3}
10^{4}
Frequency (rad/s)

```
/*******************************************************************************
* balance_config.h
*
* Contains the settings for configuration of balance.c
*******************************************************************************/

#ifndef BALANCE_CONFIG
#define BALANCE_CONFIG

// Set loop rates
#define INNER_RATE 100 // Inner oop rate
#define OUTER_RATE 20 // Outer loop rate
#define DT_INNER 0.005 // 1/SAMPLE_RATE_HZ
#define DT_OUTER 0.05 // 1/SAMPLE_RATE_HZ

// Set hardware constants
#define CAPE_MOUNT_ANGLE 0.49 // increase if mip tends to roll forward
#define GEAR_RATIO 35.555 // Motor gear ratio
#define ENCODER_RES 60 // Encoder resolution
#define MOTOR_CHANNEL_L 3 // Left motor channel
#define MOTOR_CHANNEL_R 2 // Right motor channel
#define MOTOR_POLARITY_L 1 // Left motor polarity
#define MOTOR_POLARITY_R -1 // Right motor polarity
#define ENCODER_CHANNEL_L 3 // Left encoder channel
#define ENCODER_CHANNEL_R 2 // Right encoder channel
#define ENCODER_POLARITY_L 1 // Left encoder polarity
#define ENCODER_POLARITY_R -1 // Right encoder polarity

// inner loop controller: 100hz
#define D1_GAIN 1.0
#define D1_ORDER 2
#define D1_NUM {-4.9500,  8.8709, -3.9709}
#define D1_DEN { 1.0000, -1.4810,  0.4812}
#define D1_SAT 1
#define D1_SATURATION_TIMEOUT   0.5

// outer loop controller: 20hz
#define D2_GAIN 1.0
#define D2_ORDER 1
#define D2_NUM {1.0000, -0.9961}
#define D2_DEN {1.0000, -0.6065}
#define D2_SAT 0.3

// Arming conditions
#define TIP_ANGLE 0.85
#define START_ANGLE 0.3
#define START_DELAY 0.4
#define PICKUP_DETECTION_TIME   0.6

// Other
#define TAU 2 // Complimentary Filter time constant
#define WC 0.5 // 1/TAU
#define PRINTF_HZ 10 // printf_data rate

#endif  //BALANCE_CONFIG

```

```c
/*******************************************************************************
* File: balance.c
* Author: Parker Brown
* Date: 12/15/2017
* Course: MAE 144, Fall 2017
* Description: Balance program estimates MIP state, evaluates D1 and D2
* controllers, checks out of bounds conditions to disarm.
*******************************************************************************/

// usefulincludes is a collection of common system includes for the lazy
// This is not necessary for roboticscape projects but here for convenience
// Nice to have for TWO_PI
#include <rc_usefulincludes.h>
// main roboticscape API header
#include <roboticscape.h>
#include "balance_config.h"

// Controller arming enumerated type
typedef enum arm_state_t{
    ARMED,
    DISARMED
}arm_state_t;

// Struct for angles
typedef struct angles_t{
    float theta_a_raw[2];
    float theta_g_raw[2];
    float theta_a[2];
    float theta_g[2];
    float theta;
    float theta_error[3];
    float theta_ref;

    float phi;
    float phi_error[3];
    float phi_ref;
}angles_t;

// Struct for filters
typedef struct filter_t{
    float lp_num[2];
    float lp_den[2];
    float hp_num[2];
    float hp_den[2];
} filter_t;

// Struct for controller
typedef struct controllers_t{
    float d1_u[3];
    float d2_u[3];
    arm_state_t arm_state;
} controllers_t;

// Function declarations
void angle_mananger(); // MIP state estimation
float tfn(int order, float a[], float b[], float u[], float y[], float gain, float sat);
void d1_ctrl(); // D1 Controller
void inner_loop(); // IMU ISR func with arming checks and motor driving
void d2_ctrl(); // D2 Controller
void* outer_loop(void* ptr); // outer_thread func for D2 Controller
int arm_controller(); // Ser controller state to ARMED
int disarm_controller(); // Set contrller state to DISARMED
int start_condition(); // Start with upright condition
void zero_out(); // Zero out controllers and filters
void* printf_data(void* ptr); // printf_thread func to print data
void on_pause_pressed(); // do stuff when paused button is pressed
void on_pause_released(); // do stuff when paused button is released

// Global Variables
```

```
filter_t filter; // filter struct to hold filter coefficients
angles_t angles; // angles struct to hold theta angles
controllers_t ctrl; // d1, d2 controller struct
rc_imu_data_t data; // imu struct to hold new data

// Initialize Controller coefficients
float d1_num[] = D1_NUM;
float d1_den[] = D1_DEN;

/*******************************************************************************
* int main()
*
* balance main function contains these critical components
* - call to rc_initialize() at the beginning
* - UNINITIALIZED and DISARMED
* - Initialize DMP for interrupt
* - Start and schedule outer_thread
* - Start and schedule printf_thread
* - Initialize filters
*   - Set RUNNING and start IMU ISR
* - main while loop that checks for EXITING condition
*       - checks start condition that arms controllers
* - shutdown procedures
* - rc_cleanup() at the end
*******************************************************************************/
int main(){

    // initialize hardware first
    if(rc_initialize()){
        fprintf(stderr,"ERROR: failed to initialize rc_initialize(), are you root?\n");
        return -1;
    }

    // Set UNINITIALIZED while setting up
    rc_set_led(RED,1);
    rc_set_led(GREEN,0);
    rc_set_state(UNINITIALIZED);

    // make sure controller state starts DISARMED
    ctrl.arm_state = DISARMED;

    // Initialize imu
    rc_imu_config_t conf = rc_default_imu_config(); // imu config to defaults
    conf.dmp_sample_rate = INNER_RATE;
    // Initialize imu for dmp interrupt operation
    if(rc_initialize_imu_dmp(&data, conf)){
        printf("rc_initialize_imu_failed\n");
        return -1;
    }

    // initialize pause functions
    rc_set_pause_pressed_func(&on_pause_pressed);
    rc_set_pause_released_func(&on_pause_released);

    // Check min/max sched_priority
    printf("Valid priority range for SCHED_FIFO: %d - %d\n",
                 sched_get_priority_min(SCHED_FIFO),
                 sched_get_priority_max(SCHED_FIFO));

    // Start printf_thread
    pthread_t outer_thread;
    struct sched_param outer_params;
    outer_params.sched_priority = 60; // Reasonably low priority
    pthread_create(&outer_thread, NULL, outer_loop, (void*) NULL);
    pthread_setschedparam(outer_thread, SCHED_FIFO, &outer_params);

    // Start printf_thread
    pthread_t printf_thread;
    struct sched_param printf_params;
    printf_params.sched_priority = 20; // Reasonably low priority
    pthread_create(&printf_thread, NULL, printf_data, (void*) NULL);
    pthread_setschedparam(printf_thread, SCHED_FIFO, &printf_params);

    // Initialize Low Pass filter variables
    filter.lp_num[0] = 0;
    filter.lp_num[1] = WC*DT_INNER;
    filter.lp_den[0] = 1;
    filter.lp_den[1] = WC*DT_INNER-1;
    // Initialize High Pass filter variables
    filter.hp_num[0] = 1;
    filter.hp_num[1] = -1;
    filter.hp_den[0] = 1;
    filter.hp_den[1] = WC*DT_INNER-1;

    // done initializing so set state to RUNNING
    rc_set_state(RUNNING);
    rc_set_led(GREEN, ON);  // GREEN when running
    rc_set_led(RED, OFF);  // RED when paused

    rc_set_imu_interrupt_func(&inner_loop); // IMU ISR for D1 Controller

  // Keep looping until state changes to EXITING
    while(rc_get_state()!=EXITING){
        rc_usleep(1000000 / 100); // 100hz

        // nothing to do if paused, go back to beginning of loop
        if(rc_get_state() != RUNNING) continue;

        // Wait for start condition (upright) to pass, then arm controllers
        if(ctrl.arm_state == DISARMED){
            if(start_condition()==0){
                zero_out();
                arm_controller();
            }
            else continue; // do nothing if start condition fails
        }

    }

    // Shutdown procedures
    printf("Joining printf_thread... ");
    pthread_join(printf_thread, NULL);
    printf("joined.\n");
    printf("Joining outer_thread... ");
    pthread_join(outer_thread, NULL);
    printf("joined.\n");
    disarm_controller(); // Disarm controller after closing all threads
    rc_power_off_imu();

    // exit cleanly
    rc_cleanup();
    return 0;
}

/********************************************************************************
* float tfn(int order, float a[], float b[], float u[], float y[], float gain, float sat)
*
* Computes difference equation for nth order transfer function.
* Input: denominator and numerator coefficients a[] and b[], old inputs uk's
* u[], old outputs yk's y[], tf order, tf gain, and saturation value.
* Output: float y_new from evaluation of tf.
********************************************************************************/
float tfn(int order, float a[], float b[], float u[], float y[], float gain, float sat){
    float y_new;
    y[0] = 0;
    // Assume a nd b are normalized by a[0]
    // Compute y_new for numerator coefficients b[i]
    int i;
  for(i = 0; i <= order; i++){
    y[0] += b[i] * u[i];
  }
  // Compute y_new for denominator coefficients a[j]
    int j;
  for(j = 1; j <= order; j++){
    y[0] -= a[j] * y[j];
  }

    // scale by gain
    y[0] = y[0] * gain;

    // Saturate output
    if (y[0] > sat){
        y[0] = sat;
    }   else if (y[0] < -sat){
        y[0] = -sat;
    }

    // Update values
    int k;
    for(k = order; k > 0; k--){
        u[k] = u[k-1];
        y[k] = y[k-1];
    }

    y_new = y[0];
  return y_new;
}

/*******************************************************************************
* void angle_mananger()
*
* MIP state estimation, theta, phi, and their outputs calculated.
*******************************************************************************/
void angle_mananger(){
    float wheelAngleL = 0;
    float wheelAngleR = 0;

    // Complimentary Filter start
    angles.theta_a_raw[0] = -1.0 * atan2(data.accel[2], data.accel[1]); // theta [rad]
    angles.theta_g_raw[0] = angles.theta_g_raw[0] \
                                                + (data.gyro[0] * DT_INNER *
                                                DEG_TO_RAD); // theta [rad]

    // Run high and low pass filter
    tfn(1, filter.lp_den, filter.lp_num, angles.theta_a_raw, angles.theta_a, 1, 100);
  tfn(1, filter.hp_den, filter.hp_num, angles.theta_g_raw, angles.theta_g, 1, 100);

  // Sum of Low and High pass filters of theta
    angles.theta = angles.theta_a[0] + angles.theta_g[0];
    // Correct for BBBlue mount angle
    angles.theta += CAPE_MOUNT_ANGLE;
    // Complimentary Filter end

    // Theta error for D1
    angles.theta_error[0] = angles.theta_ref - angles.theta;

    // Get phi [rad] for D2 Controller
    wheelAngleL = ((rc_get_encoder_pos(ENCODER_CHANNEL_L) * TWO_PI) \
                                / (ENCODER_POLARITY_L * GEAR_RATIO * ENCODER_RES));
    wheelAngleR = ((rc_get_encoder_pos(ENCODER_CHANNEL_R) * TWO_PI) \
                                / (ENCODER_POLARITY_R * GEAR_RATIO * ENCODER_RES));
    angles.phi = ((wheelAngleL + wheelAngleR)/2) + angles.theta;
    angles.phi_error[0] = angles.phi_ref - angles.phi;
}

/*******************************************************************************
* void inner_loop()
```

```c
*
* Inner (fast) loop run in interrupt service routine. Gets data, angles, errors.
* Checks tipping and loop saturation, disarms on failure. Runs D1 Controller.
* Drives motors if everyting passes.
********************************************************************************/
void inner_loop(){
    static int sat_counter = 0;
    float duty = 0;

    /**************************************************************
    * MIP state estimation: phi and theta angles
    ****************************************************************/
    angle_mananger();

    /**************************************************************
    * check for various exit conditions AFTER state estimate
    ****************************************************************/
    //DISARM if EXITING
    if(rc_get_state() == EXITING){
        rc_disable_motors();
        return;
    }
    // DISARM if not RUNNING (i.e. PAUSED)
    if((rc_get_state() != RUNNING) && (ctrl.arm_state == ARMED)){
        disarm_controller();
        return;
    }
    // Return out of loop if DISARMED
    if(ctrl.arm_state == DISARMED){
        return;
    }
    // DISARM if tip over detected
    if(fabs(angles.theta) > TIP_ANGLE){
        disarm_controller();
        printf("tip detected \n");
        return;
    }

    /**************************************************************
    * Run inner loop if checks pass.
    *************************************************************/
    // Second order tf for D1 Controller
    tfn(D1_ORDER, d1_den, d1_num, angles.theta_error, ctrl.d1_u, D1_GAIN, D1_SAT);

    /**************************************************************
    * Check if the inner loop saturated. If it saturates for over
    * the timout, DISARM the controller.
    *************************************************************/
    if(fabs(ctrl.d1_u[0]) > 0.95) sat_counter++;
    else sat_counter = 0;
    // if saturate for a second, disarm for safety
    if(sat_counter > (INNER_RATE * D1_SATURATION_TIMEOUT)){
        printf("inner loop controller saturated\n");
        disarm_controller();
        sat_counter = 0;
        return;
    }

    /**********************************************************
    * Drive motors.
    ***********************************************************/
    duty = ctrl.d1_u[0]; // Set duty cycle to write to motors
    rc_set_motor(MOTOR_CHANNEL_L, MOTOR_POLARITY_L * duty);
    rc_set_motor(MOTOR_CHANNEL_R, MOTOR_POLARITY_R * duty);

    return;
}

/********************************************************************************
```

```c
* void outer_loop()
*
* Runs D2 controller in outer_loop thread.
*******************************************************************************/
void* outer_loop(void* ptr){
    float d2_num[] = D2_NUM;
    float d2_den[] = D2_DEN;
    while(rc_get_state()!=EXITING){
        // Just run D2 Controller and wait
        // Second order tf for D2 Controller
        tfn(D2_ORDER, d2_den, d2_num, angles.phi_error, ctrl.d2_u, D2_GAIN, D2_SAT);
        angles.theta_ref = ctrl.d2_u[0]; // theta ref passed to inner controller
        rc_usleep(1000000 / OUTER_RATE); // Sleep to set outer loop rate
    }
    return NULL;
}

/*******************************************************************************
* int disarm_controller()
*
* Disable motors and set arming state to DISARMED
*******************************************************************************/
int disarm_controller(){
    rc_disable_motors();
    ctrl.arm_state = DISARMED;
    return 0;
}

/*******************************************************************************
* int arm_controller()
*
* Zero out the controllers and encoders. Enable motors and arm the controllers.
*******************************************************************************/
int arm_controller(){
    zero_out();
    rc_set_encoder_pos(ENCODER_CHANNEL_L,0);
    rc_set_encoder_pos(ENCODER_CHANNEL_R,0);
    ctrl.arm_state = ARMED;
    rc_enable_motors();
    return 0;
}

/*******************************************************************************
* int start_condition()
*
* Wait for MiP to be held upright long enough to initiate arming.
* Returns -1 on fail.
*******************************************************************************/
int start_condition(){
    int count = 0;
    const int count_hz = 20;    // check 20 times per second
    int count_needed = round(START_DELAY*count_hz);
    int wait_us = 1000000/count_hz;

    // Wait for MIP to be tipped out of START_ANGLE range
    while(rc_get_state() == RUNNING){
        // if within range, start counting
        if(fabs(angles.theta) > START_ANGLE) count++;
        // fell out of range, restart counter
        else count = 0;
        // waited long enough, return
        if(count >= count_needed) break;
        rc_usleep(wait_us);
    }
    // Wait for MIP to be within START_ANGLE range
    count = 0;
    while(rc_get_state() == RUNNING){
        // If within range, start counting
        if(fabs(angles.theta) < START_ANGLE) count++;
        // Else out of range and restart count
        else count = 0;
        // Return if waited long enough
        if(count >= count_needed) return 0;
        rc_usleep(wait_us);
    }
    return -1;
}

/*******************************************************************************
* void zero_out()
*
* Zero out filter inputs nad integration values.
*******************************************************************************/
void zero_out(){
    // Complimentary filter values
    angles.theta_a_raw[0] = 0;
    angles.theta_a_raw[1] = 0;
    angles.theta_g_raw[0] = 0;
    angles.theta_g_raw[1] = 0;
    angles.theta_a[0] = 0;
    angles.theta_a[1] = 0;
    angles.theta_g[0] = 0;
    angles.theta_g[1] = 0;
    // D1 Controller feedback
    angles.theta = 0;
    // D1 Controller inputs
    angles.theta_ref = 0;
    angles.theta_error[0] = 0;
    angles.theta_error[1] = 0;
    angles.theta_error[2] = 0;
    // D1 Controller outputs
    ctrl.d1_u[0] = 0;
    ctrl.d1_u[1] = 0;
    ctrl.d1_u[2] = 0;
    // D2 Controller feedback
    angles.phi = 0;
    // D2 Controller inputs
    angles.phi_ref = 0;
    angles.phi_error[0] = 0;
    angles.phi_error[1] = 0;
    angles.phi_error[2] = 0;
    // D2 Controller outputs
    ctrl.d2_u[0] = 0;
    ctrl.d2_u[1] = 0;
    ctrl.d2_u[2] = 0;
}

/*******************************************************************************
* void* printf_data(void* ptr)
*
* printf_thread function prints data.
*******************************************************************************/
void* printf_data(void* ptr){
    rc_state_t last_rc_state, new_rc_state; // keep track of last state
    last_rc_state = rc_get_state();
    while(rc_get_state()!=EXITING){
        new_rc_state = rc_get_state();
        // First time in RUNNING, print header
        if((new_rc_state == RUNNING) && (last_rc_state != RUNNING)){
            printf("\nRUNNING: Hold upright to balance.\n|");
            printf("    θ    |");
            printf("  θ_ref  |");
            printf("    φ    |");
            printf("  φ_ref  |");
            printf("  d1_u   |");
            printf("  d2_u   |");
            printf(" theta_a |");
            printf(" theta_g |");
            printf("arm_state|");
            printf("\n");
        }
        else if(new_rc_state==PAUSED && last_rc_state!=PAUSED){
            // First time being PAUSED, print pause statement
            printf("\nPAUSED: Press pause again to start.\n");
        }
        last_rc_state = new_rc_state; // update last_rc_state

        // Print data while RUNNING
        if(new_rc_state == RUNNING){
            // Print raw angles
            printf("\r|"); // carriage return because it looks pretty
            printf(" %7.3f |", angles.theta);
            printf(" %7.3f |", angles.theta_ref);
            printf(" %7.3f |", angles.phi);
            printf(" %7.3f |", angles.phi_ref);
            printf(" %7.3f |", ctrl.d1_u[0]);
            printf(" %7.3f |", ctrl.d2_u[0]);
            printf(" %7.3f |", angles.theta_a[0]);
            printf(" %7.3f |", angles.theta_g[0]);

            if(ctrl.arm_state == ARMED) printf("  ARMED  |");
            else printf("DISARMED |");
            fflush(stdout);
        }

        rc_usleep(1000000 / PRINTF_HZ); // Sleep to set print rate
    }
    return NULL;
}

/*******************************************************************************
* void on_pause_released()
*
* Make the Pause button toggle between paused and running states.
*******************************************************************************/
void on_pause_released(){
    // toggle betewen paused and running modes
    if(rc_get_state()==RUNNING){
        rc_set_state(PAUSED);
        disarm_controller(); // Always set DISARMED on PAUSE change
        rc_set_led(GREEN, OFF); // GREEN when running
        rc_set_led(RED, ON); // RED when paused
    }
    else if(rc_get_state()==PAUSED){
        rc_set_state(RUNNING);
        disarm_controller(); // Always set DISARMED on PAUSE change
        rc_set_led(GREEN, ON); // GREEN when running
        rc_set_led(RED, OFF); // RED when paused
    }
    return;
}

/*******************************************************************************
* void on_pause_pressed()
*
* If the user holds the pause button for 2 seconds, set state to exiting which
* triggers the rest of the program to exit cleanly.
*******************************************************************************/
void on_pause_pressed(){
    int i=0;
    const int samples = 100;    // check for release 100 times in this period
    const int us_wait = 2000000; // 2 seconds

    // now keep checking to see if the button is still held down
    for(i=0;i<samples;i++){
        rc_usleep(us_wait/samples);
        if(rc_get_pause_button() == RELEASED) return;
    }
    printf("long press detected, shutting down\n");
    rc_set_state(EXITING);
    return;
}

```

```
# This is a general use makefile for robotics cape projects written in C.
# Just change the target name to match your main source code filename.
TARGET = balance

CC      := gcc
LINKER      := gcc -o
CFLAGS      := -c -Wall -g
LFLAGS      := -lm -lrt -lpthread -lroboticscape

SOURCES     := $(wildcard *.c)
INCLUDES    := $(wildcard *.h)
OBJECTS     := $(SOURCES:$%.c=$%.o)

prefix      := /usr/local
RM      := rm -f
INSTALL     := install -m 4755
INSTALLDIR  := install -d -m 755

LINK        := ln -s -f
LINKDIR     := /etc/roboticscape
LINKNAME    := link_to_startup_program


# linking Objects
$(TARGET): $(OBJECTS)
    @$(LINKER) $(@) $(OBJECTS) $(LFLAGS)


# compiling command
$(OBJECTS): %.o : %.c $(INCLUDES)
    @$(CC) $(CFLAGS) -c $< -o $(@)
    @echo "Compiled: "$<

all:
    $(TARGET)

debug:
    $(MAKE) $(MAKEFILE) DEBUGFLAG="-g -D DEBUG"
    @echo " "
    @echo "$(TARGET) Make Debug Complete"
    @echo " "

install:
    @$(MAKE) --no-print-directory
    @$(INSTALLDIR) $(DESTDIR)$(prefix)/bin
    @$(INSTALL) $(TARGET) $(DESTDIR)$(prefix)/bin
    @echo "$(TARGET) Install Complete"

clean:
    @$(RM) $(OBJECTS)
    @$(RM) $(TARGET)
    @echo "$(TARGET) Clean Complete"

uninstall:
    @$(RM) $(DESTDIR)$(prefix)/bin/$(TARGET)
    @echo "$(TARGET) Uninstall Complete"

runonboot:
    @$(MAKE) install --no-print-directory
    @$(LINK) $(DESTDIR)$(prefix)/bin/$(TARGET) $(LINKDIR)/$(LINKNAME)
    @echo "$(TARGET) Set to Run on Boot"

```