```
//
// This is the main program on the BBB
// It launches the DSP code on PRU 0 (slave)
// It also launches the I2S and sample period code on PRU 1 (master)
// At the end we start tclsh as a child process.
//

#include   <stdio.h>
#include   <stdlib.h>
#include   <assert.h>
#include   <stdint.h>
#include   <math.h>
#include   "prussdrv.h"
#include   "pruss_intc_mapping.h"

#include   "mio.h"
#include   "child.h"
#include   "bbbLib.h"
#include   "mem.h"
#include   "PRUlib.h"
#include   "robotLib.h"

// TRUE and FALSE

#define   TRUE   1
#define   FALSE  0

// GUI mode or not

int  GUImode = TRUE ;

// We need a global variable that wil point to the shared memory

shared_memory_t  *shared_memory ;

// Need a variable for debugging

int             debug = TRUE ;

// Need global structure to hold GUI variables

GUIvars_t   GUIvars ;

// ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
// Routine to execute PRU programs
// ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^

void execPRUprograms() {

// Initialize the PRUs

   if (debug) printf("Initializing the PRUs.\n") ;
   PRUinit() ;

// Configure PRU 0 based on GUI settings

   if (debug) printf("Configuring PRU 0 with GUI data\n") ;
   configPRU() ;

// Start the PRUs

   if (debug) printf("Start the PRUs ...\n") ;
   PRUstart() ;
```

```
   return ;
}

// *********************************
// Main program
// *********************************

int main (void) {

   FILE    *read_from, *write_to;
   char    str[STR_LEN] ;
   int     exitFlag = FALSE ;
   int     runFlag = FALSE ;

// Print a welcome statement

   if (debug) printf("\nSIUE Beaglebot Project\n") ;
   if (debug) printf("17-Jun-2016\n\n") ;

// GPIO initialization

   if (debug) printf("Initializing the GPIOs which we will use ...\n") ;
   GPIOinit() ;
   turnLED(OFF) ;

// Look to see if user wants to use the GUI

   if (GUImode) {

// Start the gui
// Two-way pipe

      start_child("tclsh", &read_from, &write_to);
      fprintf(write_to, "source ./tcl/gui.tcl \n") ;

// Get data from GUI

      while (!exitFlag) {
          if  (fgets(str, STR_LEN, read_from) != NULL) {
             getGUIvars(str) ;
             exitFlag = GUIvars.exitFlag ;
             if (!exitFlag) {
                if (!runFlag) {
                   execPRUprograms() ;
                   testRobot() ;
                   runFlag = TRUE ;
                } else {;
                   PRUstop() ;
                   execPRUprograms() ;
                   testRobot() ;
                } // end if then elseif
             } // end if
          } //end if
      } // end while

   } else {          // NO GUI!!!!

// Get the GUI string from the robot.config file
// and parse it as usual

        loadGuiVarsFromFile(str) ;
        getGUIvars(str) ;
```

```
// The string we load may have the exit flag set
// We don’ want this ...

        GUIvars.exitFlag = FALSE ;

// Load, configure, and start the PRUs

        execPRUprograms() ;

// Run the the test robot code

        testRobot() ;

   } // end if-the-else

// User wants to exit so let PRU0 know

   exitFlag = TRUE ;
   shared_memory->exitFlag = TRUE ;

// Delay for 1 sec before disabling PRUs

   if (debug) memoryDump() ;
   pauseSec(1) ;
   PRUstop() ;

   return 0;
} // end main
```

```
// We will use I2C2 which is enumerated as 1 on the BBB (silly! I know)
// SCL on P9_17 (3.3 V tolerant)
// SDA on P9_18 (3.3 V tolerant)

#define         I2C_BUS                 2

// Base address for the servo driver module

#define         I2C_ADDR                0x40

// Write buffer size

#define         BUF_SIZE                12

// PCA9685 registers

#define         PCA9685_SUBADR1         0x02
#define         PCA9685_SUBADR2         0x03
#define         PCA9685_SUBADR3         0x04
#define         PCA9685_MODE1           0x00
#define         PCA9685_MODE2           0x01
#define         PCA9685_PRESCALE        0xfe

#define         SERVO_0_ON_L            0x06
#define         SERVO_0_ON_H            0x07
#define         SERVO_0_OFF_L           0x08
#define         SERVO_0_OFF_H           0x09

#define         ALL_SERVO_ON_L          0xfa
#define         ALL_SERVO_ON_H          0xfb
#define         ALL_SERVO_OFF_L         0xfc
#define         ALL_SERVO_OFF_H         0xfd

// Useful defines

#define         TRUE    1
#define         FALSE   0

//
// Function declaration

// Set the servo pulse repetition rate

unsigned int setServoFREQ(float) ;

// Set the pulse width of one of the servo channels

unsigned int setServoPW(int, int) ;

unsigned int resetServoDriver(void) ;
```

```
// We will use I2C2 which is called 1 here (silly)
// SCL on P9_19 (3.3 V tolerant)  I2C-2 (SCL)
// SDA on P9_20 (3.3 V tolerant)  I2C-2 (SDA)

// When 1 prints some info for debugging

#define     COLOR_SENSOR_DEBUG         1
// #define     COLOR_SENSOR_DEBUG         0

// i2c_scan 1 to test to see if device is there

#define         COLOR_SENSOR_I2CBUS            1

// Base address for the TCS34725 Color Sensor

#define         COLOR_SENSOR_ADDR                   0x29

// Write buffer size

#define         BUF_SIZE        80

// Command bit

#define     CMD_BIT         0x80

// Color sensor registers

#define     ENABLE          0x00
#define     ATIME           0x01
#define     WTIME           0x03
#define     AILTL           0x04
#define     AILTH           0x05
#define     AIHTL           0x06
#define     AIHTH           0x07
#define     PERS            0x0c
#define     CONFIG          0x0d
#define     CONTROL         0x0f
#define     ID              0x12
#define     STATUS          0x13
#define     CDATA           0x14
#define     CDATAH          0x15
#define     RDATA           0x16
#define     RDATAH          0x17
#define     GDATA           0x18
#define     GDATAH          0x19
#define     BDATA           0x1a
#define     BDATAH          0x1b

// Gain settings

#define     GAIN_1X         0x00
#define     GAIN_4X         0x01
#define     GAIN_16X        0x02
#define     GAIN_60X        0x03

// Integration time (154 ms)

#define    INTEG_TIME       0xc0

// Function declaration
// Dumps raw data from the sensor

int    init_color_sensor(void) ;
```

```
void   cleanup_color_sensor(int i2c_color_handle) ;
void   read_color_sensor(int i2c_color_handle, unsigned *c,
                        unsigned int *r, unsigned int *g, unsigned int *b) ;
```

```
// Need access to std i/o routines

#include <stdio.h>

// Need access to i2c library funcitions

#include "bbbLib.h"
#include "accelerometer.h"

// ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
// Routine to initialize the TCS3475 color sensor
// ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^

int init_accel(void){

// Set up a read buffer

   unsigned char rd_buf[BUF_SIZE] ;

// Set up a write buffer

   unsigned char wr_buf[BUF_SIZE] ;

// Get a handle for the accelerometer

   int   accel_handle ;
   accel_handle = i2c_open(ACCEL_I2C_BUS, ACCEL_I2C_ADDR);

// Let's read the ID register
// Print it to the screen when debugging
// Check it to make sure it reads 0x2a

   wr_buf[0] = CMD_BIT | ID ;
   i2c_write_read(i2c_accel_handle, ACCEL_I2C_ADDR, wr_buf, 1, ACCEL_I2C_ADDR, rd_buf,
1) ;
   if (ACCEL_DEBUG) {
         printf("The accelerometer returned the ID: %x\n", (int) rd_buf[0]) ;
   }

// Return the i2c color sensor handle

   return i2c_accel_handle ;

}

// ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
// Routine to cleanup the TCS3475 color sensor
// ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^

void  cleanup_accel(int i2c_accel_handle) {

//   unsigned char wr_buf[BUF_SIZE] ;

// Need to disable the sensor by writing value to the ENABLE register

// Clearing the lower 2 bits should disable the sensor

/*
   wr_buf[0] = CMD_BIT | ENABLE ;
   wr_buf[1] = 0x00 ;
   i2c_write(i2c_color_handle, wr_buf, 2) ;
*/
```

```
// Close the i2c channel

   i2c_close(i2c_accel_handle) ;

   return ;
}

// ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
// Routine to read the TCS3475 color sensor
// ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^

void read_accel(int i2c_accel_handle) {

// Set up a read buffer

   unsigned char rd_buf[BUF_SIZE] ;

// Set up a write buffer

   unsigned char wr_buf[BUF_SIZE] ;

// Let’s get the "clear" data from the sensor

   wr_buf[0] = CMD_BIT | CDATA ;
   i2c_write_read(i2c_color_handle, COLOR_SENSOR_ADDR, wr_buf, 1, COLOR_SENSOR_ADDR, rd
_buf, 2) ;
   if (COLOR_SENSOR_DEBUG) {
       *c = (unsigned int) rd_buf[0] ;
       *c |= ((unsigned int) rd_buf[1]) << 8 ;
       printf("\nClear data value: %u => %u %u \n", *c,
              (unsigned int) rd_buf[1],  (unsigned int) rd_buf[0]) ;
   }

// Let’s get the "red" data from the sensor

   wr_buf[0] = CMD_BIT | RDATA ;
   i2c_write_read(i2c_color_handle, COLOR_SENSOR_ADDR, wr_buf, 1, COLOR_SENSOR_ADDR, rd
_buf, 2) ;
   if (COLOR_SENSOR_DEBUG) {
       *r = (unsigned int) rd_buf[0] ;
       *r |= ((unsigned int) rd_buf[1]) << 8 ;
       printf("Red data value: %u\n", *r) ;
   }

// Let’s get the "green" data from the sensor

   wr_buf[0] = CMD_BIT | GDATA ;
   i2c_write_read(i2c_color_handle, COLOR_SENSOR_ADDR, wr_buf, 1, COLOR_SENSOR_ADDR, rd
_buf, 2) ;
   if (COLOR_SENSOR_DEBUG) {
       *g = (unsigned int) rd_buf[0] ;
       *g |= ((unsigned int) rd_buf[1]) << 8 ;
       printf("Green data value: %u\n", *g) ;
   }
// Let’s get the "blue" data from the sensor

   wr_buf[0] = CMD_BIT | BDATA ;
   i2c_write_read(i2c_color_handle, COLOR_SENSOR_ADDR, wr_buf, 1, COLOR_SENSOR_ADDR, rd
_buf, 2) ;
   if (COLOR_SENSOR_DEBUG) {
       *b = (unsigned int) rd_buf[0] ;
       *b |= ((unsigned int) rd_buf[1]) << 8 ;
       printf("Blue data value: %u\n", *b) ;
```

```
    }

// Exit

    return ;
}
```

```
/*
//Filename: libBBB.h
//Version : 0.1
//
//Project : Argonne National Lab - Forest
//Author  : Gavin Strunk
//Contact : gavin.strunk@gmail.com
//Date    : 28 June 2013
//
//Description - This is the main header file for
//              the libBBB library.
//
//Revision History
//      0.1:  Wrote basic framework with function
//              prototypes and definitions. \GS
*/

/*
Copyright (C) 2013 Gavin Strunk

Permission is hereby granted, free of charge, to any person obtaining a copy of this so
ftware and associated documentation files (the "Software"), to deal in the Software wit
hout restriction, including without limitation the rights to use, copy, modify, merge,
publish, distribute, sublicense, and/or sell copies of the Software, and to permit pers
ons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies o
r substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INC
LUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR P
URPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABL
E FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR
OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER
DEALINGS IN THE SOFTWARE.
*/


#ifndef _libBBB_H_
#define _libBBB_H_

//Includes

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>
#include <termios.h>
#include <fcntl.h>
#include <unistd.h>
#include <linux/i2c-dev.h>
#include <sys/ioctl.h>
#include <linux/i2c.h>
#include <errno.h>
#include <time.h>

//
// Set TTY to which ever UART you plan to use
//

#define TTY     "/dev/ttyO1"

// Define a UART structure
```

```
//Type definitions
typedef struct {
      struct termios u;
}UART;


//Definitions

#define OUT     "out"
#define IN      "in"
#define ON      1
#define OFF     0
#define USR1    "usr1"
#define USR2    "usr2"
#define USR3    "usr3"
#define P8_13   "P8_13"
#define E       65
#define RS      27
#define D4      46
#define D5      47
#define D6      26
#define D7      44
#define AIN0    "/AIN0"
#define AIN1    "/AIN1"
#define AIN2    "/AIN2"
#define AIN3    "/AIN3"
#define AIN4    "/AIN4"
#define AIN5    "/AIN5"
#define AIN6    "/AIN6"
#define AIN7    "/AIN7"

//Device Tree Overlay
//int addOverlay(char *dtb, char *overname);

//USR Prototypes
int setUsrLedValue(char* led, int value);

//GPIO Prototypes
int initPin(int pinnum);
int setPinDirection(int pinnum, char* dir);
int setPinValue(int pinnum, int value);
int getPinValue(int pinnum);

//PWM Prototypes
int initPWM(int mgrnum, char* pin);
int setPWMPeriod(int helpnum, char* pin, int period);
int setPWMDuty(int helpnum, char* pin, int duty);
int setPWMOnOff(int helpnum, char* pin, int run);


//UART Prototypes
//int initUART(int mgrnum, char* uartnum);
int initUART();
void closeUART(int fd);
int configUART(UART u, int property, char* value);
int txUART(int uart, unsigned char data);
unsigned char rxUART(int uart);
int UARTputstr(int uart,  char* buf);
int UARTgetstr(int uart,  char* buf);

//I2C Prototypes
```

```
int i2c_open(unsigned char bus, unsigned char addr);

int i2c_write(int handle, unsigned char* buf, unsigned int length);

int i2c_read(int handle, unsigned char* buf, unsigned int length);

int i2c_write_read(int handle,
                   unsigned char addr_w, unsigned char *buf_w, unsigned int len_w,
                   unsigned char addr_r, unsigned char *buf_r, unsigned int len_r);

int i2c_write_ignore_nack(int handle,
                          unsigned char addr_w, unsigned char* buf, unsigned int length
);

int i2c_read_no_ack(int handle,
                    unsigned char addr_r, unsigned char* buf, unsigned int length);

int i2c_write_byte(int handle, unsigned char val);

int i2c_read_byte(int handle, unsigned char* val);

int i2c_close(int handle);

// Provides an inaccurate delay (may be useful for waiting for ADC etc).
// The maximum delay is 999msec

int delay_ms(unsigned int msec);


//SPI Prototypes
int initSPI(int modnum);
void closeSPI(int device);
int writeByteSPI(int device,unsigned char *data);
int writeBufferSPI(int device, unsigned char *buf, int len);
int readByteSPI(int device, unsigned char *data);
int readBufferSPI(int device, int numbytes, unsigned char *buf);

//LCD 4-bit Prototypes
int initLCD();
int writeChar(unsigned char data);
int writeCMD(unsigned char cmd);
int writeString(char* str, int len);
int LCD_ClearScreen();
int LCD_Home();
int LCD_CR();
int LCD_Backspace();
int LCD_Move(int location);

//ADC Prototypes
int initADC(int mgrnum);
int readADC(int helpnum, char* ach);

//Time Prototypes
void pauseSec(int sec);
int  pauseNanoSec(long nano);

#endif
```

```
/*
//Filename: libBBB.c
//Version : 0.1
//
//Project : Argonne National Lab - Forest
//Author  : Gavin Strunk
//Contact : gavin.strunk@gmail.com
//Date    : 28 June 2013
//
//Description - This is the main library file that
//              eases the interface to the BeagleBone
//              Black. It includes functions for GPIO,
//              UART, I2C, SPI, ADC, Timing, and Overlays.
//
//Revision History
//      0.1:  Wrote the basic framework for all the
//              functions. \GS
*/

/*
Copyright (C) 2013 Gavin Strunk

Permission is hereby granted, free of charge, to any person obtaining a copy of this so
ftware and associated documentation files (the "Software"), to deal in the Software wit
hout restriction, including without limitation the rights to use, copy, modify, merge,
publish, distribute, sublicense, and/or sell copies of the Software, and to permit pers
ons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies o
r substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INC
LUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR P
URPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABL
E FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR
OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER
DEALINGS IN THE SOFTWARE.
*/


#include "bbbLib.h"

//Local functions not used by outside world

void initCMD(unsigned char cmd);



//***************************************************
//*                  USR FUNCTIONS                  *
//***************************************************
int setUsrLedValue(char* led, int value)
{
        FILE *usr;
        char buf[20];
        char buf2[50] = "/sys/class/leds/beaglebone:green:";

        //build file path to usr led brightness
        sprintf(buf,"%s",led);
        strcat(buf2,strcat(buf,"/brightness"));

        usr = fopen(buf2, "w");
        if(usr == NULL) printf("USR Led failed to open\n");
```

```
        fseek(usr,0,SEEK_SET);
        fprintf(usr,"%d",value);
        fflush(usr);
        fclose(usr);

        return 0;
}

//**********************************************
//*               GPIO FUNCTIONS               *
//**********************************************
int initPin(int pinnum)
{
        FILE *io;

        io = fopen("/sys/class/gpio/export", "w");
        if(io == NULL) printf("Pin failed to initialize\n");
        fseek(io,0,SEEK_SET);
        fprintf(io,"%d",pinnum);
        fflush(io);
        fclose(io);

        return 0;
}

int setPinDirection(int pinnum, char* dir)
{
        FILE *pdir;
        char buf[10];
        char buf2[50] = "/sys/class/gpio/gpio";

        //build file path to the direction file
        sprintf(buf,"%i",pinnum);
        strcat(buf2,strcat(buf,"/direction"));

        pdir = fopen(buf2, "w");
        if(pdir == NULL) printf("Direction failed to open\n");
        fseek(pdir,0,SEEK_SET);
        fprintf(pdir,"%s",dir);
        fflush(pdir);
        fclose(pdir);

        return 0;
}

int setPinValue(int pinnum, int value)
{
        FILE *val;
        char buf[5];
        char buf2[50] = "/sys/class/gpio/gpio";

        //build path to value file
        sprintf(buf,"%i",pinnum);
        strcat(buf2,strcat(buf,"/value"));

        val = fopen(buf2, "w");
        if(val == NULL) printf("Value failed to open\n");
        fseek(val,0,SEEK_SET);
        fprintf(val,"%d",value);
        fflush(val);
        fclose(val);

        return 0;
```

```
}

int getPinValue(int pinnum)
{
        FILE *val;
        int value;
        char buf[5];
        char buf2[50] = "/sys/class/gpio/gpio";

        //build file path to value file
        sprintf(buf,"%i",pinnum);
        strcat(buf2,strcat(buf,"/value"));

        val = fopen(buf2, "r");
        if(val == NULL) printf("Input value failed to open\n");
        fseek(val,0,SEEK_SET);
        fscanf(val,"%d",&value);
        fclose(val);

        return value;
}

//************************************************
//*                PWM FUNCTIONS                 *
//************************************************
int initPWM(int mgrnum, char* pin)
{
        FILE *pwm;
        char buf[5];
        char buf2[50] = "/sys/devices/bone_capemgr.";
        char buf3[20] = "bone_pwm_";

        //build file paths
        sprintf(buf,"%i",mgrnum);
        strcat(buf2,strcat(buf,"/slots"));

        strcat(buf3,pin);

        pwm = fopen(buf2, "w");
        if(pwm == NULL) printf("PWM failed to initialize\n");
        fseek(pwm,0,SEEK_SET);
        fprintf(pwm,"am33xx_pwm");
        fflush(pwm);
        fprintf(pwm,"%s",buf3);
        fflush(pwm);
        fclose(pwm);

        return 0;
}

int setPWMPeriod(int helpnum, char* pin, int period)
{
        FILE *pwm;
        char buf[5];
        char buf2[60] = "/sys/devices/ocp.2/pwm_test_";

        //build file path
        sprintf(buf,"%i",helpnum);
        printf("%s\n",pin);
        strcat(buf2,pin);
        strcat(buf2,".");
        strcat(buf2,strcat(buf,"/period"));
```

```
        printf("%s\n",buf2);
        pwm = fopen(buf2, "w");
        if(pwm == NULL) printf("PWM Period failed to open\n");
        fseek(pwm,0,SEEK_SET);
        fprintf(pwm,"%d",period);
        fflush(pwm);
        fclose(pwm);

        return 0;
}

int setPWMDuty(int helpnum, char* pin, int duty)
{
        FILE *pwm;
        char buf[5];
        char buf2[50] = "/sys/devices/ocp.2/pwm_test_";

        //build file path
        sprintf(buf,"%i",helpnum);
        strcat(buf2,pin);
        strcat(buf2,".");
        strcat(buf2,strcat(buf,"/duty"));

        pwm = fopen(buf2, "w");
        if(pwm == NULL) printf("PWM Duty failed to open\n");
        fseek(pwm,0,SEEK_SET);
        fprintf(pwm,"%d",duty);
        fflush(pwm);
        fclose(pwm);

        return 0;
}

int setPWMOnOff(int helpnum, char* pin, int run)
{
        FILE *pwm;
        char buf[5];
        char buf2[50] = "/sys/devices/ocp.2/pwm_test_";

        //build file path
        sprintf(buf,"%i",helpnum);
        strcat(buf2,pin);
        strcat(buf2,".");
        strcat(buf2,strcat(buf,"/run"));

        pwm = fopen(buf2, "w");
        if(pwm == NULL) printf("PWM Run failed to open\n");
        fseek(pwm,0,SEEK_SET);
        fprintf(pwm,"%d",run);
        fflush(pwm);
        fclose(pwm);

        return 0;
}

//**************************************************
//*                UART FUNCTIONS                  *
//**************************************************
int initUART()
{
        //return the int reference to the port
        struct termios old;
        struct termios new;
```

```
        int fd;

        fd = open(TTY, O_RDWR | O_NOCTTY);
        if(fd < 0)
        {
                printf("Port failed to open\n");
                return fd;
        }

        tcgetattr(fd,&old);
        bzero(&new, sizeof(new));

        new.c_cflag = B4800 | CS8 | CLOCAL | CREAD;
        new.c_iflag = IGNPAR | ICRNL;
        new.c_oflag = 0;
        new.c_lflag = 0;

        new.c_cc[VTIME] = 0;
        new.c_cc[VMIN]  = 1;

        //clean the line and set the attributes
        tcflush(fd,TCIFLUSH);
        tcsetattr(fd,TCSANOW,&new);

        return fd;
}

void closeUART(int fd)
{
        close(fd);
}

int configUART(UART u, int property, char* value)
{
        //This is used to set the configuration values
        //for the uart module


        return 0;
}

int txUART(int uart, unsigned char data)
{
        //write a single byte

        write(uart,&data,1);
        tcdrain(uart);

        return 0;
}

unsigned char rxUART(int uart)
{
        //read in a single byte
        unsigned char data;

        read(uart,&data,1);
        return data;
}

int UARTputstr(int uart,  char* buf)
{
```

```
        int i;

        for(i=0; i < strlen(buf); i++)
                txUART(uart,buf[i]);

        return 0;
}

int UARTgetstr(int uart,  char* buf)
{
    int i ;

    i = 0 ;
   while (1) {
        buf[i] = rxUART(uart) ;
        if (buf[i] == '\n') break ;
        i += 1 ;
    }
    i += 1 ;
    buf[i] = '\x0'  ;

    return 0 ;
}



//***************************************************
//*                I2C FUNCTIONS                    *
//***************************************************
// Returns a handle for i2c device at "addr" on bus "bus"

int i2c_open(unsigned char bus, unsigned char addr)
{
  int file;
  char filename[16];
  sprintf(filename,"/dev/i2c-%d", bus);

  if ((file = open(filename,O_RDWR)) < 0)
  {
    fprintf(stderr, "i2c_open open error: %s\n", strerror(errno));
    return(file);
  }
  if (ioctl(file,I2C_SLAVE,addr) < 0)
  {
    fprintf(stderr, "i2c_open ioctl error: %s\n", strerror(errno));
    return(-1);
  }
  return(file);
}

// Write out a data buffer to i2c device

int i2c_write(int handle, unsigned char* buf, unsigned int length)
{
  if (write(handle, buf, length) != length)
  {
    fprintf(stderr, "i2c_write error: %s\n", strerror(errno));
    return(-1);
  }
  return(length);
}

//  Write out a single byte to i2c device
```

```
int i2c_write_byte(int handle, unsigned char val)
{
  if (write(handle, &val, 1) != 1)
  {
    fprintf(stderr, "i2c_write_byte error: %s\n", strerror(errno));
    return(-1);
  }
  return(1);
}

// Read a specified number of bytes from i2c device

int i2c_read(int handle, unsigned char* buf, unsigned int length)
{
  if (read(handle, buf, length) != length)
  {
    fprintf(stderr, "i2c_read error: %s\n", strerror(errno));
    return(-1);
  }
  return(length);
}

// Read a single byte from the device

int i2c_read_byte(int handle, unsigned char* val)
{
  if (read(handle, val, 1) != 1)
  {
    fprintf(stderr, "i2c_read_byte error: %s\n", strerror(errno));
    return(-1);
  }
  return(1);
}

// Close the handle to the device

int i2c_close(int handle)
{
  if ((close(handle)) != 0)
  {
    fprintf(stderr, "i2c_close error: %s\n", strerror(errno));
    return(-1);
  }
  return(0);
}

// Write and read

int i2c_write_read(int handle,
                   unsigned char addr_w, unsigned char *buf_w, unsigned int len_w,
                   unsigned char addr_r, unsigned char *buf_r, unsigned int len_r)
{
        struct i2c_rdwr_ioctl_data msgset;
        struct i2c_msg msgs[2];

        msgs[0].addr=addr_w;
        msgs[0].len=len_w;
        msgs[0].flags=0;
        msgs[0].buf=buf_w;

        msgs[1].addr=addr_r;
        msgs[1].len=len_r;
```

```
        msgs[1].flags=1;
        msgs[1].buf=buf_r;

        msgset.nmsgs=2;
        msgset.msgs=msgs;

        if (ioctl(handle,I2C_RDWR,(unsigned long)&msgset)<0)
  {
                fprintf(stderr, "i2c_write_read error: %s\n",strerror(errno));
    return -1;
  }
  return(len_r);
}

// Write and ignore NACK

int i2c_write_ignore_nack(int handle,
                          unsigned char addr_w, unsigned char* buf, unsigned int length
)
{
        struct i2c_rdwr_ioctl_data msgset;
        struct i2c_msg msgs[1];

        msgs[0].addr=addr_w;
        msgs[0].len=length;
        msgs[0].flags=0 | I2C_M_IGNORE_NAK;
        msgs[0].buf=buf;

        msgset.nmsgs=1;
        msgset.msgs=msgs;

        if (ioctl(handle,I2C_RDWR,(unsigned long)&msgset)<0)
  {
                fprintf(stderr, "i2c_write_ignore_nack error: %s\n",strerror(errno));
    return -1;
  }
  return(length);
}

// Read and ignore no ACK

int i2c_read_no_ack(int handle,
                    unsigned char addr_r, unsigned char* buf, unsigned int length)
{
        struct i2c_rdwr_ioctl_data msgset;
        struct i2c_msg msgs[1];

        msgs[0].addr=addr_r;
        msgs[0].len=length;
        msgs[0].flags=I2C_M_RD | I2C_M_NO_RD_ACK;
        msgs[0].buf=buf;

        msgset.nmsgs=1;
        msgset.msgs=msgs;

        if (ioctl(handle,I2C_RDWR,(unsigned long)&msgset)<0)
  {
                fprintf(stderr, "i2c_read_no_ack error: %s\n",strerror(errno));
    return -1;
  }
  return(length);
}
```

```
// Delay for specified number of msec

int delay_ms(unsigned int msec)
{
  int ret;
  struct timespec a;
  if (msec>999)
  {
    fprintf(stderr, "delay_ms error: delay value needs to be less than 999\n");
    msec=999;
  }
  a.tv_nsec=((long)(msec))*1E6d;
  a.tv_sec=0;
  if ((ret = nanosleep(&a, NULL)) != 0)
  {
    fprintf(stderr, "delay_ms error: %s\n", strerror(errno));
  }
  return(0);
}

//***********************************************
//*                LCD FUNCTIONS                *
//***********************************************
/*NOTE: DO NOT directly include libBBB.h for LCD functions!
 *      Instead include libLCD.h as this implements the
 *      screen control and full initialization.
*/
int initLCD()
{
        //initialize the pins
        initPin(RS);
        initPin(E);
        initPin(D4);
        initPin(D5);
        initPin(D6);
        initPin(D7);

        //set direction
        setPinDirection(RS,OUT);
        setPinDirection(E,OUT);
        setPinDirection(D4,OUT);
        setPinDirection(D5,OUT);
        setPinDirection(D6,OUT);
        setPinDirection(D7,OUT);

        setPinValue(E,OFF);

        //initialize the screen
        pauseNanoSec(1500000);
        initCMD(0x30);
        pauseNanoSec(5000000);
        initCMD(0x30);
        pauseNanoSec(5000000);
        initCMD(0x30);
        pauseNanoSec(5000000);
        initCMD(0x20);

        pauseNanoSec(5000000);
        writeCMD(0x2C);
        pauseNanoSec(5000000);
        writeCMD(0x08);
        pauseNanoSec(5000000);
        writeCMD(0x01);
```

```
        pauseNanoSec(2000000);
        writeCMD(0x06);
        pauseNanoSec(5000000);
        writeCMD(0x0E);
        pauseNanoSec(5000000);

        return 0;
}

void initCMD(unsigned char cmd)
{
        //bring rs low for command
        setPinValue(RS,OFF);
        pauseNanoSec(500000);

        //send the highest nibble only
        setPinValue(E,ON);
        setPinValue(D7,((cmd >> 7) & 1));
        setPinValue(D6,((cmd >> 6) & 1));
        setPinValue(D5,((cmd >> 5) & 1));
        setPinValue(D4,((cmd >> 4) & 1));
        pauseNanoSec(500000);
        setPinValue(E,OFF);
        pauseNanoSec(500000);
}

int writeChar(unsigned char data)
{
        //bring rs high for character
        pauseNanoSec(500000);
        setPinValue(RS,ON);
        pauseNanoSec(500000);

        //send highest nibble first
        setPinValue(E,ON);
        setPinValue(D7, ((data >> 7) & 1));
        setPinValue(D6, ((data >> 6) & 1));
        setPinValue(D5, ((data >> 5) & 1));
        setPinValue(D4, ((data >> 4) & 1));
        pauseNanoSec(500000);
        setPinValue(E,OFF);
        pauseNanoSec(500000);

        //send the low nibble
        setPinValue(E,ON);
        setPinValue(D7, ((data >> 3) & 1));
        setPinValue(D6, ((data >> 2) & 1));
        setPinValue(D5, ((data >> 1) & 1));
        setPinValue(D4, (data & 1));
        pauseNanoSec(500000);
        setPinValue(E,OFF);
        pauseNanoSec(500000);

        return 0;
}

int writeCMD(unsigned char cmd)
{
        //bring rs low for command
        setPinValue(RS, OFF);
        pauseNanoSec(500000);

        //send highest nibble first
```

```
        setPinValue(E,ON);
        setPinValue(D7, ((cmd >> 7) & 1));
        setPinValue(D6, ((cmd >> 6) & 1));
        setPinValue(D5, ((cmd >> 5) & 1));
        setPinValue(D4, ((cmd >> 4) & 1));
        pauseNanoSec(500000);
        setPinValue(E,OFF);
        pauseNanoSec(500000);

        //send the low nibble
        setPinValue(E,ON);
        setPinValue(D7, ((cmd >> 3) & 1));
        setPinValue(D6, ((cmd >> 2) & 1));
        setPinValue(D5, ((cmd >> 1) & 1));
        setPinValue(D4, (cmd & 1));
        pauseNanoSec(500000);
        setPinValue(E, OFF);
        pauseNanoSec(500000);

        return 0;
}


//**************************************************
//*                ADC FUNCTIONS                   *
//**************************************************
int initADC(int mgrnum)
{
        FILE *ain;
        char buf[5];
        char buf2[50] = "/sys/devices/bone_capemgr.";

        //build path to setup ain
        sprintf(buf,"%i",mgrnum);
        strcat(buf2,strcat(buf,"/slots"));

        ain = fopen(buf2, "w");
        if(ain == NULL) printf("Analog failed load\n");
        fseek(ain,0,SEEK_SET);
        fprintf(ain,"cape-bone-iio");
        fflush(ain);
        fclose(ain);

        return 0;
}

int readADC(int helpnum, char* ach)
{
        FILE *aval;
        int value;
        char buf[5];
        char buf2[50] = "/sys/devices/ocp.2/helper.";

        //build file path to read adc
        sprintf(buf,"%i",helpnum);
        strcat(buf2,strcat(buf,ach));

        aval = fopen(buf2, "r");
        if(aval == NULL) printf("Analog failed to open\n");
        fseek(aval,0,SEEK_SET);
        fscanf(aval,"%d",&value);
        fflush(aval);
        fclose(aval);
```

```c
        return value;
}

//**********************************************
//*             TIME FUNCTIONS                 *
//**********************************************
void pauseSec(int sec)
{
        time_t now,later;

        now = time(NULL);
        later = time(NULL);

        while((later - now) < (double)sec)
                later = time(NULL);
}

int pauseNanoSec(long nano)
{
        struct timespec tmr1,tmr2;

        //assume you are not trying to pause more than 1s
        tmr1.tv_sec = 0;
        tmr1.tv_nsec = nano;

        if(nanosleep(&tmr1, &tmr2) < 0)
        {
                printf("Nano second pause failed\n");
                return -1;
        }
        return 0;
}
```

```
// Need access to std i/o routines

#include <stdio.h>

// Need access to i2c library funcitions

#include "bbbLib.h"

// Need access to routine for SRF02 sonar module

#include "srf02.h"

// Routine to get time from real time clock

unsigned int get_srf02_range(void) {

   int                                  i2c_srf02_handle ;
   unsigned short int   MSbyte, LSbyte ;
   unsigned int                 range ;

// Set up a read buffer

   unsigned char rd_buf[BUF_SIZE] ;

// Set up a write buffer

   unsigned char wr_buf[BUF_SIZE] ;

// Get a handle for the I2C SRF02 sonar sensor

   i2c_srf02_handle = i2c_open(I2CBUS, ADDR);

// We need to send the acquire command
// $51 will get us a range in cm
// The command needs to be registered to register 0x00

   wr_buf[0] = 0x00 ;
   wr_buf[1] = 0x51 ;

// We will write a register location and command (2 bytes) to the SRF02

   i2c_write(i2c_srf02_handle, wr_buf, 2) ;

// Need to wait 70 ms (maximum time for the echo to return)

   delay_ms(70) ;

// Now read the range (2 bytes)
// Write the register location we want first
// We need to read from registers $02 and $03

   wr_buf[0] = 0x02 ;
   i2c_write_read(i2c_srf02_handle, ADDR, wr_buf, 1, ADDR, rd_buf, 2) ;

// Convert the 2 bytes to range

   LSbyte = (unsigned int) (rd_buf[1]) ;
   MSbyte = (unsigned int) (rd_buf[0]) ;
   range = LSbyte + (MSbyte << 8) ;

// Close the i2c channel

   i2c_close(i2c_srf02_handle) ;
```

```
// Exit

   return(range);
}
```

```
//
// PRU related routines
//
// ********************************
// Initialization routine
// ********************************

#include "prussdrv.h"
#include "pruss_intc_mapping.h"

#include <stdio.h>
#include <stdint.h>
#include <stdlib.h>
#include <assert.h>
#include <math.h>

#include <unistd.h>
#include <stdlib.h>
#include <sys/types.h>

#include "mem.h"
#include "PRUlib.h"

// Global variable that points to shared memory

extern  shared_memory_t  *shared_memory ;

// Debug variable

extern  int   debug ;

// ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
// PRU initialization routine
// ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^

void PRUinit(void) {
// Initialize structure used by prussdrv_pruintc_intc
// PRUSS_INTC_INITDATA is found in pruss_intc_mapping.h

   tpruss_intc_initdata pruss_intc_initdata = PRUSS_INTC_INITDATA;

/* Allocate and initialize memory */

   prussdrv_init ();

// For PRU 0

   prussdrv_open (PRU_EVTOUT_0);

// For PRU 1

   prussdrv_open (PRU_EVTOUT_1);

// Map PRU’s INTC

   prussdrv_pruintc_init(&pruss_intc_initdata);

// Set up pointer to shared memory

   static void *pruSharedDataMemory;
   prussdrv_map_prumem(PRUSS0_SHARED_DATARAM, &pruSharedDataMemory);
   shared_memory = (shared_memory_t *) pruSharedDataMemory;
```

```
   return ;
}

// *****************************************
// Routine to clean up after the PRUs
// *****************************************

void PRUstop(void) {

/* Disable PRU and close memory mappings */

   if (debug) printf("Disabling the PRUs\n") ;
   prussdrv_pru_disable(PRU0);
   prussdrv_pru_disable(PRU1);
   prussdrv_exit ();
   return ;
}

// ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
// Here is a subroutine to interact with the PRUs
// ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^

void PRUstart(void) {

// Load and execute binary on PRU0
// Since using C-code for PRU, we need to give START_ADDR

   if (debug) printf("Starting PRU0 program\n") ;
   prussdrv_exec_program_at(PRU0, "./text.bin", START_ADDR);

/* Load and execute binary on PRU1 */

   if (debug) printf("Starting PRU1 program\n") ;
   prussdrv_exec_program(PRU1, "./pru1.bin");

   return ;
}
```

```
// We will use I2C2 which is called 1 here (silly)
// SCL on P9_19 (3.3 V tolerant)  I2C-2 (SCL)
// SDA on P9_20 (3.3 V tolerant)  I2C-2 (SDA)

// When 1 prints some info for debugging

#define     ACCEL_DEBUG          1

// i2c_scan 1 to test to see if device is there

#define         ACCEL_I2C_BUS            1

// Base address for the accelerometer

#define         ACCEL_I2C_ADDR              0x1d

// Write buffer size

#define         BUF_SIZE        80

// Command bit

#define     CMD_BIT         0x80

// Register addresses

#define     STATUS     0x00
#define     ID         0x0d


// Function declaration
// Dumps raw data from the sensor
```

```
/* memory mapped ios */


#include <stdio.h>
#include <fcntl.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/mman.h>
#include "mio.h"


int mio_open(mio_handle_t* mio, size_t off, size_t size)
{
  static const size_t page_size = 0x1000;
  static const size_t page_mask = page_size - 1;
  size_t x;
  int fd;

  fd = open("/dev/mem", O_RDWR | O_SYNC);
  if (fd == -1) return -1;

  /* align on page size */

  x = off & page_mask;
  if (x)
  {
    mio->off = x;
    off -= x;
    size += x;
  }
  else
  {
    mio->off = 0;
  }

  x = size & page_mask;
  if (x) size += page_size - x;

  mio->size = size;

  mio->base = (uintptr_t)
    mmap(NULL, mio->size, PROT_READ | PROT_WRITE, MAP_SHARED, fd, off);

  close(fd);

  if (mio->base == (uintptr_t)MAP_FAILED) return -1;

  return 0;
}

int mio_close(mio_handle_t* mio)
{
  munmap((void*)mio->base, mio->size);
  return 0;
}

uint32_t mio_read_uint32(mio_handle_t* mio, size_t off)
{
  /* off the offset in bytes */
  const size_t mio_off = mio->off + off;
  return ((volatile uint32_t*)mio->base)[mio_off / sizeof(uint32_t)];
}
```

```
void mio_write_uint32(mio_handle_t* mio, size_t off, uint32_t x)
{
  /* off the offset in bytes */
  const size_t mio_off = mio->off + off;
  ((volatile uint32_t*)mio->base)[mio_off / sizeof(uint32_t)] = x;
}

void mio_and_uint32(mio_handle_t* mio, size_t off, uint32_t x)
{
  const uint32_t xx = mio_read_uint32(mio, off);
  mio_write_uint32(mio, off, xx & x);
}

void mio_or_uint32(mio_handle_t* mio, size_t off, uint32_t x)
{
  const uint32_t xx = mio_read_uint32(mio, off);
  mio_write_uint32(mio, off, xx | x);
}

uint16_t mio_read_uint16(mio_handle_t* mio, size_t off)
{
  /* off the offset in bytes */
  const size_t mio_off = mio->off + off;
  return ((volatile uint16_t*)mio->base)[mio_off / sizeof(uint16_t)];
}

void mio_write_uint16(mio_handle_t* mio, size_t off, uint16_t x)
{
  /* off the offset in bytes */
  const size_t mio_off = mio->off + off;
  ((volatile uint16_t*)mio->base)[mio_off / sizeof(uint16_t)] = x;
}

void mio_and_uint16(mio_handle_t* mio, size_t off, uint16_t x)
{
  const uint16_t xx = mio_read_uint16(mio, off);
  mio_write_uint16(mio, off, xx & x);
}

void mio_or_uint16(mio_handle_t* mio, size_t off, uint16_t x)
{
  const uint16_t xx = mio_read_uint16(mio, off);
  mio_write_uint16(mio, off, xx | x);
}
```

```
//
// Here is a set of subrotuines useful for the robot
//

#include    <stdio.h>
#include    <stdlib.h>
#include    <stdint.h>
#include    <math.h>
#include    <string.h>

#include   "bbbLib.h"
#include   "fix.h"
#include   "mem.h"
#include   "srf02.h"
#include   "servo_driver.h"
#include   "robotLib.h"

// Global variable
// Pointer to shared memory

  extern shared_memory_t  *shared_memory ;

// GUI variables

  extern  GUIvars_t   GUIvars ;

// Debug variable

  extern  int   debug ;

// ************************************************
// Routine to initialize the GPIO we need
// ************************************************
void GPIOinit(void) {

   initPin(ACCEL_PIN) ;                    // GPIO0[2] Accel GPIO interrupt
   setPinDirection(ACCEL_PIN, IN) ;

   initPin(GPIO_LED_PIN) ;                 // GPIO1[12] GPIO LED
   setPinDirection(GPIO_LED_PIN, OUT) ;

   initPin(GPIO_SW_PIN) ;                  // GPIO1[15] GPIO SWITCH
   setPinDirection(GPIO_SW_PIN, IN) ;

   initPin(DRV_PIN) ;                      // GPIO3[19] buffer enable
   setPinDirection(DRV_PIN, OUT) ;
   setPinValue(DRV_PIN, ON) ;

   return ;
} // end GPIOinit()


// ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
// Rotuine to read the GPIO switch
// ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
int buttonPress(void) {
   int  value ;
   value = getPinValue(GPIO_SW_PIN) ;
   return value ;
} //


// ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
```

```
// Routine to turn GPIO LED (GPIO1[12]) board on or OFF
// 0 is OFF and 1 is ON
// ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
void turnLED(int state) {
    setPinValue(GPIO_LED_PIN, state) ;
    return ;
} // end turnLED()


// ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
// Routine to get GUI variables
// ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
void getGUIvars(char *str) {
   FILE   *fid ;

// Save the str sent back from tcl script to a file

   fid = fopen("./robot.config", "w") ;
   fprintf(fid, "%s\n", str) ;
   fclose(fid) ;

// Parse the string sent back from the gui

   sscanf(str, "%d:%d:%d:%d:%d:%d:%f:%f:%f:%f:%f:%f:%f:%d:%d:%d:%d:%d",
          &GUIvars.exitFlag,
          &GUIvars.sonarEna,
          &GUIvars.lineEna,
          &GUIvars.rtcEna,
          &GUIvars.accelEna,
          &GUIvars.motorType,
          &GUIvars.Kp,
          &GUIvars.Ki,
          &GUIvars.Kd,
          &GUIvars.samplePeriod,
          &GUIvars.wheelDiam,
          &GUIvars.turnRad,
          &GUIvars.ticsPerRev,
          &GUIvars.M1Ena,
          &GUIvars.M2Ena,
          &GUIvars.M3Ena,
          &GUIvars.M4Ena,
                  &GUIvars.PWMresMode
          ) ;

// Dump to the screen if we are in debug mode

   if (debug) {
          printf("exit flag is %d\n", GUIvars.exitFlag) ;
          printf("sonarEna is %d\n", GUIvars.sonarEna) ;
          printf("lineEna is %d\n", GUIvars.lineEna) ;
          printf("rtcEna is %d\n", GUIvars.rtcEna) ;
          printf("accelEna is %d\n", GUIvars.accelEna) ;
          printf("Kp is %f\n", GUIvars.Kp) ;
          printf("Ki is %f\n", GUIvars.Ki) ;
          printf("Kd is %f\n", GUIvars.Kd) ;
          printf("samplePeriod is %f\n", GUIvars.samplePeriod) ;
          printf("wheelDiam is %f\n", GUIvars.wheelDiam) ;
          printf("turnRad is %f\n", GUIvars.turnRad) ;
          printf("ticsPerRev is %f\n", GUIvars.ticsPerRev) ;
          printf("M1Ena is %d\n", GUIvars.M1Ena) ;
          printf("M2Ena is %d\n", GUIvars.M2Ena) ;
          printf("M3Ena is %d\n", GUIvars.M3Ena) ;
          printf("M4Ena is %d\n", GUIvars.M4Ena) ;
```

```
          printf("PWMresMode is %d\n", GUIvars.PWMresMode) ;
   } // end if

   return ;
}

// #########################################################
// Routine to init configure PRUs
// #########################################################

void configPRU(void) {
   float  scr = 0.0 ;  // scratchpad

   if (debug) printf("In configPRU() \n") ;

// Tells us when to exit program from GUI mode

   shared_memory->exitFlag = GUIvars.exitFlag ;

// Not currently using delay

   shared_memory->delay = 0 ;

// PWM resolution
// Sample period is in ms

   switch (GUIvars.PWMresMode) {
      case BITS_IS_8:    shared_memory->PWMres = 255 ;
                         scr = (GUIvars.samplePeriod / PWM_CLK_PERIOD_8BIT) + 0.5 ;
                         break ;
      case BITS_IS_10:   shared_memory->PWMres = 1023 ;
                         scr = (GUIvars.samplePeriod / PWM_CLK_PERIOD_10BIT) + 0.5 ;
                         break ;
      case BITS_IS_12:   shared_memory->PWMres = 4095 ;
                         scr = (GUIvars.samplePeriod / PWM_CLK_PERIOD_12BIT) + 0.5 ;
                         break ;
   } // end switch


// Number of PWM clock cycles making up a PID sample period

   shared_memory->PWMclkCnt = (uint32_t) (scr) ;

// Either DC or Servo

   shared_memory->motorType = GUIvars.motorType ;

// Motor enables

   shared_memory->motorENA[M1] = GUIvars.M1Ena ;
   shared_memory->motorENA[M2] = GUIvars.M2Ena ;
   shared_memory->motorENA[M3] = GUIvars.M3Ena ;
   shared_memory->motorENA[M4] = GUIvars.M4Ena ;

// Wheel diameter and tics per inch

   shared_memory->wheelDiam = TOFIX(GUIvars.wheelDiam, Q) ;
   scr = GUIvars.ticsPerRev / (PI * GUIvars.wheelDiam) ;
   shared_memory->ticsPerInch = TOFIX(scr, Q) ;

// Initialize DC motor structures
// Multiply the Kp, Ki, Kd values by constant
// We do this so we can use sliders in the GUI
```

```
// which span to 0 to 100 range.

   float k ;
   k = 0.001 * GUIvars.samplePeriod ;

   int i ;
   for (i = 0; i < NUM_MOTORS; i++) {
       shared_memory->motor[i].Kp = TOFIX((k * GUIvars.Kp), Q) ;
       shared_memory->motor[i].Ki = TOFIX((k * GUIvars.Ki), Q) ;
       shared_memory->motor[i].Kd = TOFIX((k * GUIvars.Kd), Q) ;
       shared_memory->motor[i].PWMmax = shared_memory->PWMres ;
       shared_memory->motor[i].PWMmin = 0 ;
   } // end i loop

// Freeze the PRUs implementing motor control

   shared_memory->command.code = NOP ;
   shared_memory->command.status = IDLE ;
   shared_memory->state = 0 ;

   return ;
} // end configPRU()


// ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
// Routine to load robot paramters from a file
// rather than from the GUI
// ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^

void loadGuiVarsFromFile(char * str) {
   FILE  *fid ;

   fid = fopen("./robot.config", "r") ;
   fscanf(fid, "%s", str) ;
   if (debug) printf("robot.config string read -> %s\n", str) ;
   fclose(fid) ;

   return  ;
} // end loadGuiVarsFromFile()


// %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
// Dump of entire memory structure to a file
// %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

void memoryDump(void) {
   FILE  *fid ;
   int   i ;

   if (debug) printf("Dumping contents of shared memory to file.\n") ;

   fid = fopen("memory_dump.txt", "w") ;

// Motor enable values

   for (i = 0; i < NUM_MOTORS; i++) {
        fprintf(fid, "mem->motorENA[%d] = %d\n", i+1, shared_memory->motorENA[i]) ;

   } // end i loop

// PWM and encoder values

   for (i = 0; i < NUM_MOTORS; i++) {
```

```
        fprintf(fid, "mem->pwm[%d] = %d\n", i+1, shared_memory->pwm[i]) ;
        fprintf(fid, "mem->enc[%d] = %d\n", i+1, shared_memory->enc[i]) ;
   } // end i loop

// Motor parameters

   for (i = 0; i < NUM_MOTORS; i++) {
        fprintf(fid, "mem->motor[%d].setpoint = %d\n", i+1, shared_memory->motor[i].set
point) ;
        fprintf(fid, "mem->motor[%d].distance = %d\n", i+1, shared_memory->motor[i].dis
tance) ;
        fprintf(fid, "mem->motor[%d].targetDistance = %d\n", i+1, shared_memory->motor[
i].targetDistance) ;
        fprintf(fid, "mem->motor[%d].wheelDirection = %d\n", i+1, shared_memory->motor[
i].wheelDirection) ;
        fprintf(fid, "mem->motor[%d].brakeType = %d\n", i+1, shared_memory->motor[i].br
akeType) ;
        fprintf(fid, "mem->motor[%d].e0 = %d\n", i+1, shared_memory->motor[i].e0) ;
        fprintf(fid, "mem->motor[%d].e1 = %d\n", i+1, shared_memory->motor[i].e1) ;
        fprintf(fid, "mem->motor[%d].e2 = %d\n", i+1, shared_memory->motor[i].e2) ;
        fprintf(fid, "mem->motor[%d].Kp = %d\n", i+1, shared_memory->motor[i].Kp) ;
        fprintf(fid, "mem->motor[%d].Ki = %d\n", i+1, shared_memory->motor[i].Ki) ;
        fprintf(fid, "mem->motor[%d].Kd = %d\n", i+1, shared_memory->motor[i].Kd) ;
        fprintf(fid, "mem->motor[%d].PWMmin = %d\n", i+1, shared_memory->motor[i].PWMmi
n) ;
        fprintf(fid, "mem->motor[%d].PWMmax = %d\n", i+1, shared_memory->motor[i].PWMma
x) ;
        fprintf(fid, "mem->motor[%d].PWMout = %d\n", i+1, shared_memory->motor[i].PWMou
t) ;
   } // end i loop

// Other parameters

   fprintf(fid, "mem->wheelDiam = %d\n", shared_memory->wheelDiam) ;
   fprintf(fid, "mem->ticsperInch = %d\n", shared_memory->ticsPerInch) ;
   fprintf(fid, "mem->delay = %d\n", shared_memory->delay) ;
   fprintf(fid, "mem->state = %x\n", shared_memory->state) ;
   fprintf(fid, "mem->PWMclkCnt = %d\n", shared_memory->PWMclkCnt) ;
   fprintf(fid, "mem->PWMres = %d\n", shared_memory->PWMres) ;
   fprintf(fid, "mem->exitFlag = %d\n", shared_memory->exitFlag) ;
   fprintf(fid, "mem->motorType = %u\n", shared_memory->motorType) ;
   fprintf(fid, "mem->scr = %d\n", shared_memory->scr) ;
   fprintf(fid, "mem->interruptCounter = %u\n", shared_memory->interruptCounter) ;
   fprintf(fid, "mem->command.code = %u\n", shared_memory->command.code) ;
   fprintf(fid, "mem->command.status = %u\n", shared_memory->command.status) ;

// Data buffer

/*
   for (i=0; i<BUF_LEN; i++) {
      fprintf(fid, "mem->enc_data[%d] = %u\n", i, shared_memory->enc_data[i]) ;
   }
*/

// Close the file and exit

   fclose(fid) ;
   return ;
} // end memoryDump()


// ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
// Routine to convert a distance in inches to an
```

```
// equivalent number of encoder tics.
// ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^

int32_t inches2tics(float inches) {
   int32_t   tics ;
   int32_t   inches_Q ;

   inches_Q = TOFIX(inches, Q) ;
   tics = FMUL(shared_memory->ticsPerInch, inches_Q, 0) ;
   tics = FCONV(tics, twoQ, 0) ; // integer
   return tics ;
} // end inches2tics()


// ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
// Routine to convert a distance in tics to an
// equivalent number of inches
// ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^

float tics2inches(int32_t tics) {
   float   inches ;
   inches = ((float) tics) / shared_memory->ticsPerInch ;
   return inches ;
} // end tics2inches()


// ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
// Routine to update a motor structure.
// Values are stored into motor structures.
// Target distance, setpoint, wheel direction and braking
// mode get written out to shared memory.
// The final setpoint is actually stored in targetSetpoint.
// The setpoint is always set to 0.  The move routine in
// PRU 0 will setpoint up to the target.
// ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^

void updateMotor(int motor_num, int dir, int brakeType, float distance, float velocity)
 {
   int32_t   distInTics, velInTics, deltaVel ;
   float     delX ;

// delX is distance we would move in one sample period
// samplePeriod is in ms

   delX = velocity * GUIvars.samplePeriod * 0.001 ;

   distInTics = inches2tics(distance) ;
   velInTics =  inches2tics(delX) ;

   shared_memory->motor[motor_num].targetDistance = distInTics ;
   shared_memory->motor[motor_num].targetSetpoint = velInTics ;
   deltaVel = velInTics >> 5 ;  // divide by 32
   if (deltaVel < 1) deltaVel = 1 ;
   shared_memory->motor[motor_num].deltaSetpoint = deltaVel ;
   shared_memory->motor[motor_num].setpoint = 0 ;
   shared_memory->motor[motor_num].wheelDirection = dir ;
   shared_memory->motor[motor_num].brakeType = brakeType ;

   return ;

} // end updateMotor()

// ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
```

```
// Routine to query a motor structure for important values.
// ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
int32_t queryMotor(int motor_num, int item) {
   int32_t  value ;

   switch(item) {
      case  SETPOINT:    value = shared_memory->motor[motor_num].setpoint ;
                         break ;
      case  DISTANCE  :  value = shared_memory->motor[motor_num].distance ;
                         break ;
      case  TARGET_DIST: value = shared_memory->motor[motor_num].targetDistance ;
                         break ;
      case  WHEEL_DIR:   value = shared_memory->motor[motor_num].wheelDirection ;
                         break ;
      case  BRAKE_TYPE:  value = shared_memory->motor[motor_num].brakeType ;
                         break ;
      default:           value = CRASH;
                         break ;
   } // end switch

   return value ;

} // end updateMotor()

// ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
// Routine to query the command structure
// Makes it easy to determine status of command
// ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
int32_t query(int item) {
   int32_t  value ;

   switch(item) {
      case  CMD:         value = shared_memory->command.code ;
                         break ;
      case  STATUS:      value = shared_memory->command.status ;
                         break ;
      default:           value = CRASH ;
                         break ;
   } // end switch

   return value ;

} // end updateMotor()

// ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
// Wait for IDLE to be true
// ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^

void waitForIdle(void) {
   while (query(STATUS) != IDLE) { delay_ms(STATUS_DELAY) ; }
   return ;
} // end for waitForIdle()


// ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
// Wait for Complete to be true
// ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^

void waitForComplete(void) {
   while (query(STATUS) != COMPLETED) { delay_ms(STATUS_DELAY) ; }
   return ;
} // waitForComplete()
```

```
// ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
// Reset the PRUs
// ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^

void resetPRU(void) {
   if (debug) printf("Entering resetPRU()\n") ;
   shared_memory->command.code = BRAKE_HARD ;
   shared_memory->command.status = START ;
   waitForComplete() ;
   shared_memory->command.code = NOP ;
   shared_memory->command.status = IDLE ;
   return ;
} // end resetPRU()


// ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
// Routine to drive robot forward
// a distance (in inches) with a given velocity
// (in inches per sec)
//
// Return a 1 if successfully started command
// Return a 0 if not successful
//
// Only implementing for two motors.
// ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
int fwd(float distance, float velocity) {

// Wait until we are idle

   if (debug) { printf("Waiting for IDLE in fwd().\n") ; }

   waitForIdle() ;

// Update motor structures with information about
// how forward command should be carried out

   updateMotor(M1, CW, HARD, distance, velocity) ;
   updateMotor(M2, CW, HARD, distance, velocity) ;

// Update command structure to indicate we desre to drive FWD

   shared_memory->command.code = FWD ;
   shared_memory->command.status = START ;

   if (debug) { printf("Waiting for COMPLETED in fwd().\n") ; }
   waitForComplete() ;

// After command is seen to complete then set
// command to a no-op and state as being idle

   shared_memory->command.code = NOP ;
   shared_memory->command.status = IDLE ;
   shared_memory->state = 0 ;

   return PASS ;

} // end fwd()


// ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
// Routine to drive robot backward
// Only implementing for two motors.
```

```
// M1 is left motor
// M2 motor is right
// ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^

int  bwd(float distance, float velocity) {

// Wait until we are idle

   if (debug) { printf("Waiting for IDLE in bwd();\n") ; }
   waitForIdle() ;

// Update motor structures with information about
// how forward command should be carried out

   updateMotor(M1, CCW, HARD, distance, velocity) ;
   updateMotor(M2, CCW, HARD, distance, velocity) ;

// Update command structure to indicate we desre to drive BWD

   shared_memory->command.code = BWD ;
   shared_memory->command.status = START ;

   if (debug) { printf("Waiting for COMPLETED in bwd() ;\n") ; }
   waitForComplete() ;

// After command is seen to complete then set
// command to a no-op and state as being idle

   shared_memory->command.code = NOP ;
   shared_memory->command.status = IDLE ;
   shared_memory->state = 0 ;

   return PASS ;

} // end bwd()


// ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
// Routine to spin robot in direction specified
// # of degrees at particular velocity.
// Only implementing for two motors.
// One motor gets driven CW and the other CCW.
// ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
int rotate(float degrees, float velocity, int direction) {
   float  distance ;
//
// Need to determin distance we need to travel
//
   distance = (PI / 180.0) * GUIvars.turnRad * degrees ;

// Wait until we are idle

   if (debug) { printf("Waiting for IDLE in rotate().\n") ; }
   waitForIdle() ;

// Update motor structures with information about
// how forward command should be carried out

   if (direction == CW) {
      updateMotor(M1, CCW, HARD, distance, velocity) ;
      updateMotor(M2, CW, HARD, distance, velocity) ;

   } else {
```

```
      updateMotor(M1, CW, HARD, distance, velocity) ;
      updateMotor(M2, CCW, HARD, distance, velocity) ;
   } // end if-then-else

// Update command structure to indicate we desre to ROT

   shared_memory->command.code = ROT ;
   shared_memory->command.status = START ;

   if (debug) { printf("Waiting for COMPLETED in rotate().\n") ; }
   waitForComplete() ;

// After command is seen to complete then set
// command to a no-op and state as being idle

   shared_memory->command.code = NOP ;
   shared_memory->command.status = IDLE ;
   shared_memory->state = 0 ;

   return PASS ;
} // end rotate() ;


// ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
// Routine to drive to make a right turn
// Just calling the rotate() routine.
// Only implementing for two motors.
// Turn at 6 in/sec
// ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
int right(void) {

   if (rotate(90.0, 6.0, CW) == PASS) {
      return PASS ;
   } else {
      return FAIL ;
   } // end if-then-else

} // end right()

// ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
// Routine to drive to make a left turn
// Just calling the rotate() routine.
// Only implementing for two motors.
// Turn at 6 in/sec
// ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
int left(void) {

   if (rotate(90.0, 6.0, CCW) == PASS) {
      return PASS ;
   } else {
      return FAIL ;
   } // end if-then-else

} // end left()

// ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
// Routine to take a test drive
// Drive forward 24 inches at 6 in/sec
// ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
void testDrive(void) {

// Reset PRU .. hard brake
```

```
   resetPRU() ;

// Drive forward (36 inches at 6 in/sec)

   fwd(36.0, 12.0) ;

/*
// Drive backward

   bwd(20.0, 4.0) ;

// Right turn

   right() ;

// Left turn

   left() ;
*/
   return  ;
} // end test_drive()

// ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
// Routine to test the sonar unit
// ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
void testSonar() {
/*
    unsigned int  range ;

// Get range from srf02

   range = get_srf02_range() ;

// Print the range

   printf("\nRange ==> %d cm\n\n", range);
*/
   return  ;
} // end test_sonar()

// ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
// Routine to test the servo driver
// ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
void testServo(void) {
/*
   int i ;

// Reset the servo driver

   resetServoDriver() ;

// Set rep rate to 50 Hz

   setServoFREQ(50.0) ;

// Turn servo one direction and then back the other

   for (i=150; i<=350; i+=50) {
       setServoPW(0, i) ;
       delay_ms(900) ;
       delay_ms(900) ;
    }
*/
```

```
   return ;
} // end test_servo()

// ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
// Routine to test our robot
// ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^

void testRobot(void) {

   if (debug) printf("Entering testRobot()\n") ;

   testDrive() ;

   return ;
}
```

```
// We will use I2C2 which is called 1 here (silly)
// SCL on P9_19 (3.3 V tolerant)  I2C-2 (SCL)
// SDA on P9_20 (3.3 V tolerant)  I2C-2 (SDA)

// i2c_scan 1 to test to see if device is there

#define         I2CBUS          1

// Base address for the SRF02 sonar module

#define         ADDR            0x70

// Write buffer size

#define         BUF_SIZE        12

// Function declaration
// Gets range from the srf02

unsigned int get_srf02_range(void) ;
```

```
// Need access to std i/o routines

#include <stdio.h>

// Need access to i2c library funcitions

#include "bbbLib.h"
#include "color_sensor.h"

// ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
// Routine to initialize the TCS3475 color sensor
// ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^

int init_color_sensor(void){

// Set up a read buffer

   unsigned char rd_buf[BUF_SIZE] ;

// Set up a write buffer

   unsigned char wr_buf[BUF_SIZE] ;

// Get a handle for the TCS3475 color sensor

   int   i2c_color_handle ;
   i2c_color_handle = i2c_open(COLOR_SENSOR_I2CBUS, COLOR_SENSOR_ADDR);

// Need to enable the sensor by writing value to the ENABLE register

   wr_buf[0] = CMD_BIT | ENABLE ;

// Setting the lower to bits should enable the sensor

   wr_buf[1] = 0x03 ;
   i2c_write(i2c_color_handle, wr_buf, 2) ;

// Let’s read the ID register
// Print it to the screen when debugging
// Check it to make sure it reads 0x44

   wr_buf[0] = CMD_BIT | ID ;
   i2c_write_read(i2c_color_handle, COLOR_SENSOR_ADDR, wr_buf, 1, COLOR_SENSOR_ADDR, rd
_buf, 1) ;
   if (COLOR_SENSOR_DEBUG) {
         printf("The color sensor returned the ID: %x\n", (int) rd_buf[0]) ;
   }

// Let’s set the gain of the sensor

   wr_buf[0] = CMD_BIT | CONTROL ;
   wr_buf[1] = GAIN_16X ;
   i2c_write(i2c_color_handle, wr_buf, 2) ;

   if (COLOR_SENSOR_DEBUG) {
         printf("Gain setting is: %x\n", (int) wr_buf[1]) ;
   }

// Let’s set the integration time of the sensor

   wr_buf[0] = CMD_BIT | ATIME ;
   wr_buf[1] = INTEG_TIME ;
   i2c_write(i2c_color_handle, wr_buf, 2) ;
```

```
   if (COLOR_SENSOR_DEBUG) {
         printf("Integration time is: %x\n", (int) wr_buf[1]) ;
   }

// Return the i2c color sensor handle

   return i2c_color_handle ;

}

// ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
// Routine to cleanup the TCS3475 color sensor
// ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^

void  cleanup_color_sensor(int i2c_color_handle) {

//   unsigned char wr_buf[BUF_SIZE] ;

// Need to disable the sensor by writing value to the ENABLE register

// Clearing the lower 2 bits should disable the sensor

/*
   wr_buf[0] = CMD_BIT | ENABLE ;
   wr_buf[1] = 0x00 ;
   i2c_write(i2c_color_handle, wr_buf, 2) ;
*/

// Close the i2c channel

   i2c_close(i2c_color_handle) ;

   return ;
}

// ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
// Routine to read the TCS3475 color sensor
// ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^

void read_color_sensor(int i2c_color_handle, unsigned int *c,
                       unsigned int *r, unsigned int *g, unsigned int *b) {

// Set up a read buffer

   unsigned char rd_buf[BUF_SIZE] ;

// Set up a write buffer

   unsigned char wr_buf[BUF_SIZE] ;

// Let’s get the "clear" data from the sensor

   wr_buf[0] = CMD_BIT | CDATA ;
   i2c_write_read(i2c_color_handle, COLOR_SENSOR_ADDR, wr_buf, 1, COLOR_SENSOR_ADDR, rd
_buf, 2) ;
   if (COLOR_SENSOR_DEBUG) {
       *c = (unsigned int) rd_buf[0] ;
       *c |= ((unsigned int) rd_buf[1]) << 8 ;
       printf("\nClear data value: %u => %u %u \n", *c,
              (unsigned int) rd_buf[1],  (unsigned int) rd_buf[0]) ;
   }
```

```
// Let’s get the "red" data from the sensor

   wr_buf[0] = CMD_BIT | RDATA ;
   i2c_write_read(i2c_color_handle, COLOR_SENSOR_ADDR, wr_buf, 1, COLOR_SENSOR_ADDR, rd
_buf, 2) ;
   if (COLOR_SENSOR_DEBUG) {
       *r = (unsigned int) rd_buf[0] ;
       *r |= ((unsigned int) rd_buf[1]) << 8 ;
       printf("Red data value: %u\n", *r) ;
   }

// Let’s get the "green" data from the sensor

   wr_buf[0] = CMD_BIT | GDATA ;
   i2c_write_read(i2c_color_handle, COLOR_SENSOR_ADDR, wr_buf, 1, COLOR_SENSOR_ADDR, rd
_buf, 2) ;
   if (COLOR_SENSOR_DEBUG) {
       *g = (unsigned int) rd_buf[0] ;
       *g |= ((unsigned int) rd_buf[1]) << 8 ;
       printf("Green data value: %u\n", *g) ;
   }
// Let’s get the "blue" data from the sensor

   wr_buf[0] = CMD_BIT | BDATA ;
   i2c_write_read(i2c_color_handle, COLOR_SENSOR_ADDR, wr_buf, 1, COLOR_SENSOR_ADDR, rd
_buf, 2) ;
   if (COLOR_SENSOR_DEBUG) {
       *b = (unsigned int) rd_buf[0] ;
       *b |= ((unsigned int) rd_buf[1]) << 8 ;
       printf("Blue data value: %u\n", *b) ;
   }

// Exit

   return ;
}
```

```
//^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
// Defines
//^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^

// BUF_LEN is length of data buffer
// STR_LEN is length of string buffer

#define     BUF_LEN         200
#define     STR_LEN         250

// Motor defines

#define     NUM_MOTORS      4

#define     M1              0
#define     M2              1
#define     M3              2
#define     M4              3

#define     CRASH           -9999

// Wheel directions

#define   CW      0
#define   CCW     1

// Brake types

#define   COAST   0
#define   HARD    1
//
// Commands we can give PRU 0
//

#define   NOP            0
#define   FWD            1
#define   BWD            2
#define   ROT            3
#define   BRAKE_HARD     4
#define   BRAKE_COAST    5
#define   HALT_PRU       6

//
// Here the codes for status of commands
//

#define   CMD            1
#define   STATUS         2

#define   IDLE          0
#define   START         1
#define   ACTIVE        2
#define   COMPLETED     3
#define   ABORTED       4

// These are used when we query the motor structure

#define   SETPOINT       1
#define   DISTANCE       2
#define   TARGET_DIST    3
#define   WHEEL_DIR      4
#define   BRAKE_TYPE     5
```

```
// ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
// A structure that decribes a command from
// the ARM to PRU 0
// ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^

typedef struct {
    int32_t     code ;
    int32_t     status ;
} command_t ;


// ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
// Declare a structure to hold the GUI variables
// ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^

typedef struct {
    int      exitFlag ;
    int      sonarEna ;
    int      lineEna ;
    int      rtcEna ;
    int      accelEna ;
    int      motorType ;
    float    Kp ;
    float    Ki ;
    float    Kd ;
    float    samplePeriod ;
    float    wheelDiam ;
    float    turnRad ;
    float    ticsPerRev ;
    int      M1Ena ;
    int      M2Ena ;
    int      M3Ena ;
    int      M4Ena ;
    int      PWMresMode ;
} GUIvars_t ;


// ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
// A DC motor structure
// ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^

typedef struct {
    int32_t     setpoint ;          // desired velocity (in tics)
    int32_t     targetSetpoint ;    // will rammp up intil this is reached
    int32_t     deltaSetpoint ;     // steps we will take in ramping up
    int32_t     distance ;          // dist in tics (actual)
    int32_t     targetDistance ;    // dist in tics (desired)
    int32_t     wheelDirection ;    // CW or CCW
    int32_t     brakeType ;         // COAST or HARD
    int32_t     e0 ;                // current error
    int32_t     e1 ;                // past error
    int32_t     e2 ;                // past "past error"
    int32_t     Kp ;                // proportional gain (Q)
    int32_t     Ki ;                // integral gain (Q)
    int32_t     Kd ;                // deriviative gain (Q)
    int32_t     PWMmin ;            // minumum PWM out allowed
    int32_t     PWMmax ;            // maximum PWM out allowed
    int32_t     PWMout ;            // PWM output
}   DCmotor_t;

// ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
// Our shared memory structure
// ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
```

```
typedef struct {
    int32_t     pwm[NUM_MOTORS] ;      // shared mem byte os of 0
    int32_t     enc[NUM_MOTORS] ;      // os of 16
    int32_t     delay ;                // os of 32
    int32_t     state ;                // os of 36
    int32_t     PWMclkCnt ;            // os of 40
    int32_t     PWMres ;               // os of 44
    int32_t     exitFlag ;             // exit when true
    int32_t     interruptCounter ;     // sample counter
    int32_t     motorType ;            // DC or stepper
    int32_t     motorENA[NUM_MOTORS] ; // Motor enables
    int32_t     scr ;                  // scratchpad register
    int32_t     wheelDiam ;            // diameter in inches (Q)
    int32_t     ticsPerInch;           // encoder tics per inch (Q)
    int32_t     enc_data[BUF_LEN] ;    // Buffer of encoder data
    command_t   command ;              // Motor command structure
    DCmotor_t   motor[NUM_MOTORS] ;    // DC motor structure
}   shared_memory_t ;
```

```
//
// Functions described in PRUlib

#define     PRU0    0
#define     PRU1    1

// Function declarations

void PRUinit(void) ;
void PRUstop(void) ;
void PRUstart(void) ;
```

```
//
// This is a program to test out a series of submodules
// that might be useful for robotics
//

#define   ON       1
#define   OFF      0

// We need stdio support


#include  <stdio.h>

// And our robotic library

// #include "robotLib.h"

// And the BBB library

#include "bbbLib.h"

// And the accelerometer include file

#include "accelerometer.h"


int main(void) {

   char  str[80] ;

// Initialize the accelerometer

    int   i2c_color_handle ;
    i2c_color_handle = init_color_sensor() ;

// Read the color sensor

   unsigned int  c, r, g, b ;

   int   do_it = 1 ;
   while (do_it) {
       printf("\nRead the color sensor? (y/n)  ") ;
       if (fgets(str, 80, stdin) > 0) {
           if (str[0] == 'y') {
               read_color_sensor(i2c_color_handle, &c, &r, &g, &b) ;
           }
           else {
                do_it = 0 ;
           } // end if-then-else
       } // end if
   } // end while

// Closing up the color sensor

    cleanup_color_sensor(i2c_color_handle) ;

// Print a closing message

    if (debugPrint) printf("Robot test complete ... exiting!\n") ;
    return ;
}
```

```
#define   FALSE   0
#define   TRUE    1

#define   PASS    1
#define   FAIL    0

#define   M_RUN                  (1 << 8)
#define   M_HARD_BRAKE   (1 << 9)
#define   M_UPDATE               (1 << 10)
#define   M_HALT                 (1 << 11)


// Motor control for state register

#define   M1_CW         (0x00000004)
#define   M1_CCW        (0x00000008)
#define   M2_CW         (0x00000010)
#define   M2_CCW        (0x00000020)
#define   M3_CW         (0x00000001)
#define   M3_CCW        (0x00000002)
#define   M4_CW         (0x00000040)
#define   M4_CCW        (0x00000080)

// Period of PWM clock in ms
// Measure on the scope and enter
// correct value here

#define     PWM_CLK_PERIOD_12BIT     0.8
#define     PWM_CLK_PERIOD_10BIT     0.2
#define     PWM_CLK_PERIOD_8BIT      0.05

// PWM resolution modes

#define     BITS_IS_8          1
#define     BITS_IS_10         2
#define     BITS_IS_12         3

#define     GPIO_LED_PIN      44
#define     GPIO_SW_PIN       47
#define     ACCEL_PIN          2
#define     DRV_PIN          115

// Delay (in ms) to wait between status checks

#define     STATUS_DELAY      50

// Function prototype declarations

void    getGUIvars(char *str) ;
void    loadGuiVarsFromFile(char *str) ;
void    GPIOinit(void) ;
void    turnLED(int state) ;
int     buttonPress(void) ;
void    configPRU(void) ;
void    memoryDump(void) ;

int32_t inches2tics(float inches) ;
float   tics2inches(int32_t tics) ;
void    updateMotor(int motor_num, int dir, int brakeType, float distance, float veloci
ty) ;
int32_t queryMotor(int motor_num, int item) ;
int32_t query(int item) ;
```

```
void    resetPRU(void) ;
void    waitForIdle(void) ;
void    waitForComplete(void) ;

int     fwd(float distance, float velocity) ;
int     bwd(float distance, float velocity) ;
int     rotate(float degrees, float velocity, int direction) ;
int     right(void) ;
int     left(void) ;

void    testDrive(void) ;
void    testSonar(void) ;
void    testServo(void) ;
void    testRobot(void) ;
```

```c
/* child.c */
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <sys/time.h>
#include "child.h"
/* Exec the named cmd as a child process, returning
 * two pipes to communicate with the process, and
 * the child’s process ID */

int start_child(char *cmd, FILE **readpipe, FILE **writepipe) {
   int childpid, pipe1[2], pipe2[2];
   if ((pipe(pipe1) < 0) || (pipe(pipe2) < 0)) {
        perror("pipe"); exit(-1);
        }
   if ((childpid = vfork()) < 0) {
     perror("fork"); exit(-1);
   } else if (childpid > 0) {  /* Parent. */
     close(pipe1[0]); close(pipe2[1]);
     /* Write to child is pipe1[1], read from
      * child is pipe2[0].  */
     *readpipe = fdopen(pipe2[0],"r");
     *writepipe=fdopen(pipe1[1],"w");
     setlinebuf(*writepipe);
     return childpid;
   } else {  /* Child. */
     close(pipe1[1]); close(pipe2[0]);
     /* Read from parent is pipe1[0], write to
      * parent is pipe2[1].  */
     dup2(pipe1[0],0);
     dup2(pipe2[1],1);
     close(pipe1[0]); close(pipe2[1]);
     if (execlp(cmd,cmd,NULL) < 0)
        perror("execlp");
     /* Never returns */
    return 0 ;
} }
```

```
// Need access to std i/o routines

#include <stdio.h>

// Need access to i2c library funcitions

#include "bbbLib.h"

// Need access to servo_driver.h

#include "servo_driver.h"

// Routine to set the PWM freqeuncy
// Routine accepts a freqeuncy between 40 Hz and 1000 Hz
// and sets the pre-scaler value in the PCA9685 servo controller

#define   DEBUG    0

unsigned int resetServoDriver(void) {

   int                   i2c_handle ;

// Set up a write buffer

   unsigned char wr_buf[BUF_SIZE] ;

// Get a handle for the I2C servo controller

   i2c_handle = i2c_open(I2C_BUS, I2C_ADDR) ;

// Write to MODE1 register

   wr_buf[0] = PCA9685_MODE1 ;
   wr_buf[1] = 0x00 ;
   i2c_write(i2c_handle, wr_buf, 2) ;

// Close the I2C channel and return

   i2c_close(i2c_handle) ;

   return(TRUE);
}

// ********************************************************************************

unsigned int setServoFREQ(float freq) {

   int                   i2c_handle ;
   float                 tmp ;
   unsigned char         pre_scale_value ;
   unsigned char         old_mode ;
   unsigned char         new_mode ;

// Set up a write buffer

   unsigned char wr_buf[BUF_SIZE] ;

// Set up a read buffer

   unsigned char rd_buf[BUF_SIZE] ;

// Make sure frequency is in the allowed range (40 Hz to 1 kHz)
// If not, then return an error
```

```
   if ((freq < 40.0) || (freq > 1000)) return(FALSE) ;

// Convert freqeuncy to a pre-scaler value

   tmp = (25.0e6 / (4096 * freq)) + 0.5 ;
   pre_scale_value = ((unsigned char) tmp)  - 1 ;
   if (DEBUG) printf("Pre-scaler value: %d\n", (int) pre_scale_value) ;

// Get a handle for the I2C servo controller

   i2c_handle = i2c_open(I2C_BUS, I2C_ADDR);

// Need to read current mode

   wr_buf[0] = PCA9685_MODE1 ;
   i2c_write_read(i2c_handle, I2C_ADDR, wr_buf, 1, I2C_ADDR, rd_buf, 1) ;
   old_mode = rd_buf[0] ;
   new_mode = (old_mode & 0x7f) | 0x10 ; // sleep mode

// Write out new mode and put PCA9685 to sleep

   wr_buf[1] = new_mode ;
   i2c_write(i2c_handle, wr_buf, 2) ;

//  Write out the prescaler value

   wr_buf[0] = PCA9685_PRESCALE ;
   wr_buf[1] = pre_scale_value ;
   i2c_write(i2c_handle, wr_buf, 2) ;

// Restore old mode setting

   wr_buf[0] = PCA9685_MODE1 ;
   wr_buf[1] = old_mode ;
   i2c_write(i2c_handle, wr_buf, 2) ;

// Wait 5 ms

   delay_ms(5) ;

// Turn on auto increment

   wr_buf[1] = old_mode | 0xa1 ;
   i2c_write(i2c_handle, wr_buf, 2) ;

// Close the I2C channel and return

   i2c_close(i2c_handle) ;

   return(TRUE);
}

// ******************************************************************
// $$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$
// ******************************************************************

// Routine to set the pulse width in ticks
// 0.5 ms to 2ms is 100 to 400

unsigned int setServoPW(int chan, int pw) {

   int             i2c_handle ;
```

```
   unsigned char   ms_byte, ls_byte ;
   unsigned char   cr_addr ;

// Set up a write buffer

   unsigned char wr_buf[BUF_SIZE] ;

// Make sure we have a valid channel number

   if ((chan < 0) || (chan > 15)) return(FALSE) ;

   if (DEBUG) printf("Channel #:  %d\n", chan) ;

// Get a handle for the I2C servo controller

   i2c_handle = i2c_open(I2C_BUS, I2C_ADDR);

// Control register base address for a particular servo
// can be computed by taking 4 * channel number + 6

   cr_addr = (unsigned char) (4 * chan + SERVO_0_ON_L);
   if (DEBUG) printf("Control register address is %d\n", (int) cr_addr) ;

// Need to break the pulse with up into two bytes

   ms_byte = (unsigned char) (pw / 256) ;
   ls_byte = (unsigned char) (pw % 256 );
   if (DEBUG) printf("ms_byte = %d, ls_byte = %d\n", (int) ms_byte, (int) ls_byte) ;

// Base address for control register

   wr_buf[0] = cr_addr ;

// Pulse turned on when counter equals $000

   wr_buf[1] = 0x00 ;
   wr_buf[2] = 0x00 ;

// Pulse turned off when counter equals ...

   wr_buf[3] = ls_byte ;
   wr_buf[4] = ms_byte ;

   i2c_write(i2c_handle, wr_buf, 5) ;

// Close the i2c channel and exit

   i2c_close(i2c_handle) ;

   return(TRUE);
}
```

```
//
// Routines that make doing fixed point operations easy
//

#define     PI        3.14159

#define     Q         6
#define     twoQ      12

#define     Q24       24
#define     Q12       12
#define     Q0        0

// Fixed point operationss

#define  FADD(op1,op2)      ( (op1) + (op2) )
#define  FSUB(op1,op2)      ( (op1) - (op2) )
#define  FMUL(op1,op2,q)    ((int32_t) (((int64_t) ((int64_t) (op1) * (int64_t) (op2)))
 >> q))

// #define  FDIV(op1,op2,q)    ( (int32_t) (((int64_t)(op1) << q)/ ((int64_t) op2 )) )

// Convert from a q1 format to q2 format

#define  FCONV(op1,q1,q2)     (((q2) > (q1)) ? ((op1) << ((q2)-(q1))) : ((op1) >> ((q1)
-(q2))))

// Convert a float to a fixed-point representation in q format

#define  TOFIX(op1, q)       ((int32_t) ((op1) * ((float) (1 << (q)))))

// Convert a fixed-point number back to a float

#define  TOFLT(op1, q)       ( ((float) (op1)) / ((float) (1 << (q))) )
```