Function and operation of the various modules of Botboardino CH3R_PS2 hexapod control software (Lynxmotion PS2 compilation) for Botboardino, SSC32 Servo Controller and the PS2 intended to control Lynxmotion hexapod robots via the PS2 controller V2

Jim Lake, Feb 2019

The Lynxmotion PS2 compilation consists of nine modules that have been stitched together in the Arduino environment. The software was written by a collaboration of authors including Jeroen Janssen and Kurt Ekhart, then and ported to the Arduino environment and modified to include Bill Porter's PS2 library. This document is intended to explain the function and operation of the various modules in order to make it a bit more accessible to the average user. The modules are:

1. Botboardio_CH3R_PS2
2. Hex_Cfg.h
3. Hex_Globals.h
4. InputController.h
5. PS2X_lib.cpp
6. PS2X_lib.h
7. PS2_controller.cpp
8. ServoDiver.h
9. Phoenix_driver_ssc32.cpp

First, the header files:

HexCfg.h – This is the hardware configuration file for the six legged robot. This file allows the user to specify certain hardware for the specific robot it's intended to be compiled for. The routine defaults to the (1) Botboardino, (2) OPT_TERMINAL_MONITOR, a terminal monitor that can be useful for debugging, (3) OPT_GPPLAYER, (4) USE_SSC32, the SSC32 servo controller.

Note: The author uses Hungarian notation where the first letter of the variable name refers to the data type where: c is constant, w is word, g_ is global, f is flag

It also:

1. defines the pin outs of Botboardino and assigns them names;
2. sets the baudrate for the SSC32 and assigns names to the SSC32 pins. This is where the servo names like cRRCoxaPin are defined;
3. Defines the minimum and maximum angles for the servos. It appears that the angles are specified in microseconds relating to the pulse widths need to turn the servo to a particular angle where: 0 is 90 deg, -1500 is 0 degrees and 1500 is 180 degrees;
4. Defines the dimensions for the legs of the robot in millimeters. The dimensions for the Phoenix hexapod are Coxa = 29, Femur = 57 and Tibia = 141;

5. Defines the body dimensions of the robot including the default setup angles for the servos and the distance from the center of the body to the various leg parts (in millimeters);
6. Sets the starting positions for the feet. It uses the XZ plane where Z is the forward/aft axis and X is perpendicular, or right/left. The Y direction is up/down. However, the code, as written, places the legs on either side at the same coordinates, so there is something amiss there. ****Find out how this data is used*******.

Hex_Globals.h – This header file declares the main global variables and functions.

It defines:

1. Global class objects, where an object is an instance of a global class. There are three of these objects, g_servoDriver, g_InputController, and g_InControlState;
2. Leg numbers, with leg 0 being the right rear leg and increasing counterclockwise as viewed from above;
3. Global variables, however most of these are removed with an #if 0. It appears that most of these are also defined in InputController.h to be part of the InControlState class.
4. It requires the following to be defined in the control program:
```
extern void     InitController(void);
extern void     ControlInput(void);
extern void     AllowControllerInterrupts(boolean fAllow);
```

InputController.h – This is the class definition for the controller class that will run the robot. The author has tried to build in flexibility with how the robot is to be controlled. It defines the data structure for the controls to manipulate. Three classes are defined here:

InputController,   No object is defined for this class here;

_Coord3D, An object called COORD3D is defined

_InControlState, an object called INCONTROLSTATE is defined

ServoDriver.h – This file defines the ServoDriver class which is used to talk to the servos. The author states that there may be several instances of the class. There are no objects of the class defined here. Recall that an object of this class is defined in Hex_Globals.h as g_ServoDriver.

PS2X_lib.h – This is Bill Porter's PS2 library. Button constants are defined here along with the PS2X class, but no class instances.

Phoenix_driver_ssc32.cpp – This is the control program for the control of the servos. This routine defines the functions pertaining to the SSC32, it does not actually call any of those functions. Meaningful code begins with the definition of the function:

OutputServoInfoForLeg(byte LegIndex, short sCoxaAngle1, short sFemurAngle1, short sTibiaAngle1)

This function defines a word variable for each servo, e.g. wCoxaSSCV, which is the servo value for the coxa(hip) servo. The servo value is calculated as follows:

(CoxaAngle1 +900) * 1000/cPwmDiv+cPFConst

Where: cPwmDiv = 991 and cPFConst = 592, and 900*(1000/cPwmDiv)+cPFConst must always be 1500.

Since 1500 is the servo centerpoint, the formula above computes a displacement from the center position of the servo. If the angle is negative, the displacement is left of center, if positive, right of center. The author states that he has used a pwm/degree factor of 10.09, or a servo pulse range of 0 to 1816 for 0 to 180 degrees of movement. That's a bit different than what you would get of you used the usual range of 0 to 2000 mS for 0 to 180 degrees of movement or an 11.1 factor. There is no indication in the code for why he did this, so perhaps it was arrived at by experimentation.

The next segment sends the data for a leg to the SSC32 using the serial.print function using decimal numbers. It follows that by sending in a move time (wmoveTime) calculated elsewhere. The function uses the ASCII SSC32 command protocol set forth in the SSC32 manaul.

The function FreeServo()centers all 32 servo by sending them a P0 command.

The function SSCRead() reads data from the SSC32.

The function FindServoOffsets() reads the servo offsets from the eeprom on the SSC32, set the servo at their center point and wiggles them the amount of the offset. It allows the user to set new offsets and save them to the SSC32.

PS2X_lib.cpp – Contains the functions necessary to read the control information from the PS2 controller. It defines a series of command sequences native to the PS2 controller that have been obtained by snooping the controller data interface. The interface is essentially a slave SPI operating in synchronous mode. It must use a 2.4 GHz radio link to transmit the data, but that link is transparent to the user. How the data is encoded on the RF link is unknown, but not pertinent.

The data is carried on two separate wires, DATA and COMMAND, which correspond to MISO and MOSI on a SPI. In addition it has a clock line CLK and chip select line, ATT, which is active high. Power and ground are there as well. The power line is labeled 5V, but it seems to work with 3.3V. There is also a separate power line to run the vibration motors in the controller.

This routine reads the gamepad button states and then tests the buttons to determine what modes of operation to set.

Botboardio_CH3R_PS2 –

This is the main act. This routine uses the functions defined in all of the other routines to control the robot and send commands to the SSC32. It contains the setup() function and the main loop() along with several other primary control functions.

First, it gets around the need to use floating point to calculate trig functions by creating a trig table in the microprocessor’s program memory. AVR’s are Harvard processors which have their program memory separate from their data memory. Constants can be stored in program memory, which is flash, thereby saving scarce RAM.

Separate tables for servo minima and maxima, leg lengths, body offsets and starting positions are also established in program memory using the values defined in the HexCfg.h file.

Next, variable arrays for the actual servo angles and leg positions are defined as e.g CoxaAngle1[1…6], LegPosX[1…6] etc. The LED’s are defined as LedA, LedB,LedC, and Eyes. Then a whole string of integer and short integer variables are defined. There are no floats.

Next, the two global objects that were not defined in the header files, g_InControlState and g_ServoDriver are defined.

Read initial leg positions from memory and put in array;

Make sure no single leg is selected for single leg control;

Set body positions x, y and z to zero;

Set body rotations and offsets to zero;

Set gait to 1, Balance Mode to 0, and Leg lift height to 50;

Initialize the input controller (PS2);

Set servo move time to 150;

The main loop follows:

```
                 ******begin main loop*******
                              |
                   Start millisecond timer
                              |
              Check Voltage – if low shut down
                              |
Get the control inputs from the controller and store them in variables
                              |
                 Write Outputs(does nothing)
                              |
                 Check for Single Leg Control
                              |
              Set Gait and do Gait calculations
                              |
        If in balance mode, Perform Balance Calculations
```

|

Perform forward then Inverse Kinematics(gives us servo angles)

|

Check Mechanical Limits

|

Write IK errors to leds A and C

|

Drive Servos: Make sure robot’s ON

|

Drive Servos: Calculate servo move time

|

Drive Servos: Send the move time to the SSC32

|

Drive Servos: Send the servo angles to the servo driver (phoenix_driver_ssc32.cpp) which sends the commands to the servos

|

Calculate the delay time based on the time it took to make calculations and nominal gait speed

|

Check the continue walking flag
If it’s up, wait the delay time,
If, not send the servos to neutral position

|

******end of loop*********

How does the compilation read the PS2 controller?
The first step in the process is a call to InputController.init() which is part of PS2controller.cpp. It configures the gamepad by calling ps2x_config_gamepad() which is in the ps2 library. It also sets some parameter and initializes a few variables.

The next step is a call to InputController.ControlInput(), also located in PS2Controller.cpp. It calls ps2x.read_gamepad() from PS2X_lib.cpp. It checks the time since the last read. If the time delay is more than 1500 milliseconds, it will reconfigure the gamepad again. If the time is less than a time delay settable time delay, the program waits to the end of the delay period and then sets up the SPI interface and calls _gamepad_shiftinout() which shifts in a control command sequence and shifts out the button settings and stores them in an array called PS2data[]. It then checks if the first byte of that array is 0x79, which tells the controller the gamepad is in full data return mode. If so, it calls _gamepad_shifinout() again and reads the next 12 bytes into PS2data[].Finally, it looks at the upper nibble of PS2data[1] to make sure it is a 7, indicating the gamepad is in analog mode. If not, it reconfigures the gamepad again to try to get it into analog mode.

The status of each digital button is accessed through ps2.ButtonPressed(button), which returns true if a button is pressed. The state of the analog sticks is accessed through ps2.Analog(button). It returns a byte indicating the location of the stick where 0 is full left(down), 128 is centered, and 255 is full right(up). The buttons are defined PS2X_lib.h.

How does the compilation handle analog data from the analog sticks?
This takes place initially in PS2Controller.cpp after the main program calls InputController.ControlInput(). The ControlMode is initially set to WALKMODE . Once the data is read out to the data array PS2data[], the routine checks the status of the digital buttons and if a button is pressed, it enters the ControlMode assigned to that button. If it is still in WALKMODE when it checks the analog data, it uses the analog data stored in the PS2data[] array to set the variables

InControlState.TravelLength.x
InControlState.TravelLength.z
InControlState.TravelLength.y

These variables are defined in the ControlState class as type COORD3D, which contains 3 long integers for x, y and z. Initially, however, they are merely 8 bit numbers ranging from 0 to 255 representing the stick deflection. The routine subtracts 128 from the stick data. So, if left stick left-right is, say, 220, then TravelLength.x would be 220-128= 92. X and Z are lengths and Y is a rotation. These are now signed integers where the sign represents direction. **Note**: Variables with similar names are also defined in Hex_Globals.h (e.g. TravelLengthX, BodyPosX…), however they are not used in the compilation. They are most likely artifacts from a previous version which were never cleaned up, but they were #if 0'ed out.

Once the variable TravelLength data is updated, the movement of the robot is dictated by the gait which is controlled by the function Gait(). Gait() sets the variables GaitPosX, GaitPosY, GaitPosZ, and GaitRotY which are functions of TravelLength.x , TravelLength.y and TravelLength.z. GaitPosX Y and Z are then used by the inverse kinematics routines BodyFK() and LegIK() to determine the servo rotations. The mathematical calculations involved in the inverse kinematics are very complex, and anyone using this compilation should be very relieved that someone else has done them.

How can one use a control mechanism other than the PS2 to control the robot?
Other control mechanisms were contemplated by the authors. Specifically, they expected that the robot would be controlled by data arriving by XBEE radio, since the Botboardino has an XBEE slot which uses the UART to send data to the microcontroller running the compilation. They included the file InputController.h in anticipation that there would be several instances of the control class. However, just one instance is included in the compilation, the PS2 controller.

There would appear to be two ways to add another means of control to the compilation. The first would be to create a new instance of the Control State class and a set of routines to implement that class and control the robot. The second would be to add some code to PS2Controller.cpp which adds a new Control Mode, say AUTONOMOUS_MODE or XBEE_MODE. When that button is toggled, the variables TravelLength.x,y and z would be determined by a another routine, say autonomous.cpp, which used inputs from sensors or gps, a compass or a accelerometer to set the travel length variables. This second method has the advantage of allowing the user to take control of the robot with the PS2 if it encountered problems by simply toggling the button.