BioloidCControl – User's Guide

Version 0.5 – November 2011

Author: Peter Lanius – PeterLanius@gmail.com

BioloidCControl is an alternative firmware for the Robotis Bioloid Premium Kit humanoid type robots. Its aim is to replicate the functionality of the original Robotis firmware (which is not open source), giving the user more options in terms of behaviour control. It is based on:

1. Robotis Embedded C toolkit v1.01 (for serial and Dynamixel control)
2. Robotis sample task and motion files for humanoid Type A/B/C robots (for motion pages and walking code)
3. Pololu robotics library (for ADC and Buzzer functions)

The main control loop implements a finite state machine which can receive commands via the serial port. Commands can be issued using RoboPlus Terminal either using a Zig2Serial or the serial cable. A complete command reference is provided below.

Prerequisites

1. Robotis Bioloid Premium Kit with ZigBee modules (Zig2Serial for wireless control)
2. RoboPlus software (for Terminal and RoboPlus Motion)
3. AVR Studio 5 – can be downloaded from www.atmel.com
4. Perl if you want to use your own motion file

Getting Started

1. Unpack the zip file into a suitable directory
2. Create a motion.h file (the included file is for a Type A humanoid robot) using:
   ```
   Cygwin> perl translate_motion.pl bio_prm_humanoidtypeX_en.mtn
   (where X = a, b or c)
   ```
3. Overwrite the default motion.h file with the one generated above
4. Import the project into AVR Studio 5
5. Open global.h and select the hardware configuration from the options provided
6. Open serial.h and select the serial interface (Zig2Serial or cable) that you will be using
7. Open BioiloidCControl.c and check the configuration options at the top for any adjustments you would like to make
8. Copy the code in `motionPageInit.c` to the motionPageInit() function in motion.c
9. Build the solution
10. Transfer the .hex file to the CM-510 controller using RoboPlus Terminal (you have to hold down Shift-# whilst pressing the Robot power button to start the bootloader)
11. Type 'go' in RoboPlus Terminal or cycle the power on the CM-510 once the download is complete
12. Wait for the PLAY LED to flash and then press the START button
13. Issue commands at the prompt as per the table below

Command Reference

`STOP`	Execute the exit page(s) of the current motion being performed. This command does not disable torque.
`Mxxx`	Execute the motion page with the number xxx. This will execute the sequence if the page(s) have a Next Page entry.
`SIT`	Execute the Sit Down motion page and disable torque.
`STND`	Execute the Stand motion page.
`BAL`	Execute the balance page (*balancing using gyros not yet implemented*).
`FGUP`	Get up from slip forward (robot is face down)
`BGUP`	Get up from slip backward (robot is on its back)
`WRDY`	Execute the Walk Ready motion page
`WFWD`	Walk Forward
`WBWD`	Walk Backward
`WLT`	Walk Left Turn
`WRT`	Walk Right Turn
`WLSD`	Walk Left Side
`WRSD`	Walk Right Side
`WFLT`	Walk Forward Left Turn
`WFRT`	Walk Forward Right Turn
`WFLS`	Walk Forward Left Side
`WFRS`	Walk Forward Right Side
`WBLT`	Walk Backward Left Side
`WBRT`	Walk Backward Right Side
`WAL`	Walk Avoid Left
`WAR`	Walk Avoid Right
`WBLT`	Walk Backward Left Turn
`WBRT`	Walk Backward Right Turn

Modules

BioloidCControl.c – main program

```
/*
 * BioloidCControl.c - Replacement firmware for the Robotis Bioloid CM-510 controller
 *   and Bioloid Premium Kit based humanoid robots (Type A/B/C).
 *
 * Requires a motion.h file generated by the translate_motion.pl Perl script.
 *
 * Supports all motions, including walking, gyro, DMS, buzzer, LEDs, buttons and
 * serial connection via cable and ZIG2Serial.
 *
 * Performs initialisations and then runs main control loop
 *
 */
```

global.h

```
/*
 * global.h - Basic definitions for the Robotis Bioloid CM-510 controller.
 *   contains hardware definitions and command list
 *
 */
```

clock.c

```
/*
 * clock.c - millisecond and microsecond clock
 *
 */

// return millisecond count
unsigned long millis(void);

// return microsecond count
unsigned long micros(void);
```

buzzer.c

```
/*
 * buzzer.c - Functions for controlling the buzzer on the Robotis CM-510
 *  These function use a timer1 PWM to generate the note frequencies and
 *     timer1 overflow interrupt to time the duration of the notes, so the
 *     buzzer can be playing a melody in the background while the rest of
 *     the code executes.
 *     Based on the Pololu library
 */

// Plays the specified sequence of notes.  If the play mode is
// PLAY_AUTOMATIC, the sequence of notes will play with no further
// action required by the user.  If the play mode is PLAY_CHECK,
// the user will need to call playCheck() in the main loop to initiate
// the playing of each new note in the sequence.  The play mode can
// be changed while the sequence is playing.
// This is modeled after the PLAY commands in GW-BASIC, with just a
// few differences.
//
// The notes are specified by the characters C, D, E, F, G, A, and
// B, and they are played by default as "quarter notes" with a
// length of 500 ms.  This corresponds to a tempo of 120
// beats/min.  Other durations can be specified by putting a number
// immediately after the note.  For example, C8 specifies C played as an
// eighth note, with half the duration of a quarter note. The special
// note R plays a rest (no sound).
//
// Various control characters alter the sound:
//   '>' plays the next note one octave higher
//
//   '<' plays the next note one octave lower
//
//   '+' or '#' after a note raises any note one half-step
//
//   '-' after a note lowers any note one half-step
//
//   '.' after a note "dots" it, increasing the length by
//       50%.  Each additional dot adds half as much as the
//       previous dot, so that "A.." is 1.75 times the length of
//       "A".
//
//   'O' followed by a number sets the octave (default: O4).
//
//   'T' followed by a number sets the tempo (default: T120).
//
//   'L' followed by a number sets the default note duration to
//       the type specified by the number: 4 for quarter notes, 8
//       for eighth notes, 16 for sixteenth notes, etc.
//       (default: L4)
//
//   'V' followed by a number from 0-15 sets the music volume.
//       (default: V15)
//
//   'MS' sets all subsequent notes to play staccato - each note is played
//       for 1/2 of its allotted time, followed by an equal period
//       of silence.
//
//   'ML' sets all subsequent notes to play legato - each note is played
//       for its full length.  This is the default setting.
//
//   '!' resets all persistent settings to their defaults.
//
// The following plays a c major scale up and back down:
//   play("L16 V8 cdefgab>cbagfedc");
//
void buzzer_play(const char *sequence);

// A version of play that takes a pointer to program space instead
// of RAM.  This is desirable since RAM is limited and the string
// must be in program space anyway.
void buzzer_playFromProgramSpace(const char *sequence);
```

button.c

```
/*
 * button.c - Functions and Interrupt Service Routines for controlling
 *     the five push buttons on the Robotis CM-510 controller.
 *
 *
 */
```

adc.c

```
/*
 * adc.c - Library for using the analog inputs on the Robotis CM-510
 *  controller. Reads ADC0 for battery voltage and ADC1-ADC6 from
 *     the external 5-pin ports. By default ports are assigned as follows:
 *            GyroX = CM-510 Port3 = ADC3 = PORTF3
 *            GyroY = CM-510 Port4 = ADC4 = PORTF4
 *            DMS   = CM-510 Port5 = ADC5 = PORTF5
 * Based on the Pololu library
 */

// function that reads all the sensors from the main loop
// SENSOR_READ_INTERVAL in global.h determines how often the sensors are read
// BATTERY_READ_INTERVAL in global.h determines how often the battery voltage is read
// Returns:  int flag = 0 when no new values have been read
//           int flag = 1 when new values have been read
int adc_readSensors();

// function to process the sensor data when new data become available
// detects slips (robot has fallen over forward/backward)
// and also low battery alarms at this stage
// Returns:  int flag = 0 no new command
//           int flag = 1 new command
int adc_processSensorData();

// returns the voltage of the battery in millivolts using
// 10 averaged samples.
uint16 adc_readBatteryMillivolts();

// converts the specified ADC result to cm for the DMS sensor
uint8 adc_convertDMStoCM(uint16 adcResult);

// take a single analog reading of the specified channel
uint16 adc_read(uint8 channel);

// take a single analog reading of the specified channel and return result in
millivolts
uint16 adc_readMillivolts(uint8 channel);

// take 'sample' readings of the specified channel and return the average
uint16 adc_readAverage(uint8 channel, uint16 samples);

// take 'sample' readings of the specified channel and return the average in
millivolts
uint16 adc_readAverageMillivolts(uint8 channel, uint16 samples);
```

led.c

```
/*
 * led.c - Functions for controlling the six LEDs
 *     on the Robotis CM-510 controller.
 */

// toggle specified LED
void led_toggle(uint8 ledIndex);

// switch specified LED on
void led_on(uint8 ledIndex);

// switch specified LED off
void led_off(uint8 ledIndex);
```

serial.c

```
/*
 * serial.c - Functions for the serial port PC interface on the
 * Robotis CM-510 controller.
 * Can either use serial cable or Zig2Serial via Zig-110
 *
 * Based on the embedded C library provided by Robotis
 *
 */

// Top level serial port task
// manages all requests to read from or write to the serial port
// Receives commands from the serial port and writes output (excluding printf)
// Checks the status flag provided by the ISR for operation
// Returns:  int flag = 0 when no new command has been received
//           int flag = 1 when new command has been received
int serialReceiveCommand();

// write a string to the serial port
void serial_write( unsigned char *pData, int numbyte );

// read a string from the serial port
unsigned char serial_read( unsigned char *pData, int numbyte );

// get the status of the input/output queue
int serial_get_qstate(void);
```

dynamixel.c

```
/*
 * dynamixel.c - Functions for the Dynamixel interface on the
 *   Robotis CM-510 controller.
 *
 * Based on the embedded C library provided by Robotis
 *
 */

// high level communication methods
// Ping a Dynamixel device
// returns the error bits from the status packet obtained
int dxl_ping(int id);

// Read data from the control table of a Dynamixel device
// Length 0x04, Instruction 0x02
// Parameter1 Starting address of the location where the data is to be read
// Parameter2 Length of the data to be read (one/two bytes in this case)
int dxl_read_byte(int id, int address);
int dxl_read_word(int id, int address);

// Function to write data into the control table of the Dynamixel actuator
// Length N+3 (N is the number of data to be written)
// Instruction 0x03
// Parameter1 Starting address of the location where the data is to be written
// Parameter2 1st data to be written
// Parameter3 2nd data to be written, etc.
// In this case we have 1 or 2 parameters respectively
// Returns communication status - see dxl_get_result
int dxl_write_byte(int id, int address, int value);
int dxl_write_word(int id, int address, int value);
```

```
// Supplementary functions to print communication errors (requires serial port to PC)
// Print error bit of status packet
void dxl_printErrorCode();
// Print communication result
void dxl_printCommStatus(int CommStatus);

// Function for controlling several Dynamixel actuators at the same time.
// The communication time decreases by using the Sync Write instruction
// since many instructions can be transmitted by a single instruction.
// However, you can use this instruction only when the lengths and addresses
// of the control table to be written to are the same.
// The broadcast ID (0xFE) needs to be used for transmitting.
// ID: 0xFE
// Length: (L + 1) * N + 4 (L: Data length for each Dynamixel actuator, N: The number
of Dynamixel actuators)
// Instruction: 0x83
// Parameter1 Starting address of the location where the data is to be written
// Parameter2 The length of the data to be written (L)
// Parameter3 The ID of the 1st Dynamixel actuator
// Parameter4 The 1st data for the 1st Dynamixel actuator
// Parameter5 The 2nd data for the 1st Dynamixel actuator
// ParameterL+4 The ID of the 2nd Dynamixel actuator
// ParameterL+5 The 1st data for the 2nd Dynamixel actuator
// ParameterL+6 The 2nd data for the 2nd Dynamixel actuator
// …
// NOTE: this function only allows 2 bytes of data per actuator
int dxl_sync_write_word( int NUM_ACTUATOR, int address, const uint8 ids[], int
values[] );

// Function setting goal and speed for all Dynamixel actuators at the same time
// Uses the Sync Write instruction (also see dxl_sync_write_word)
// Inputs:    NUM_ACTUATOR - number of Dynamixel servos
//                   ids - array of Dynamixel ids to write to
//                   goal - array of goal positions
//                   speed - array of moving speeds
//Returns:    commStatus
int dxl_set_goal_speed( int NUM_ACTUATOR, const uint8 ids[], uint16 goal[], uint16
speed[] );
```

pose.c

```
/*
 * pose.c - functions for assuming poses based on motion pages
 *
 */

// read in current servo positions to determine current pose.
void readCurrentPose();

// Function to wait out any existing servo movement
void waitForPoseFinish();

// Calculate servo speeds to achieve desired pose timing
// We make the following assumptions:
// AX-12 speed is 59rpm @ 12V which corresponds to 0.170s/60deg
// The AX-12 manual states this as the 'no load speed' at 12V
// We ignore the Moving Speed entry which states that 0x3FF = 114rpm
// Instead we assume that Moving Speed 0x3FF = 59rpm
void calculatePoseServoSpeeds(uint16 time);



// Moves from the current pose to the goal pose
// using calculated servo speeds and delay between steps
// to achieve the required step timing (actual play time)
// Inputs:  (uint16)  allocated step time in ms
//          (uint16)  array of goal positions for the actuators
//          (uint8)   flag = 0 don't wait for motion to finish
//                    flag = 1 wait for motion to finish and check alarms
// Returns    (int)    -1  - communication error
//                      0  - all ok
//                      1  - alarm
int moveToGoalPose(uint16 time, uint16 goal[], uint8 wait_flag);

// Assume default pose (Balance - MotionPage 224)
void moveToDefaultPose(void);
```

motion.c

```
/*
 * motion.c - functions for executing motion pages
 *     requires a motion.h file created by the translate_motion Perl script
 *
 */

// Initialize the motion pages by constructing a table of pointers to each page
// Motion pages are stored in Flash (PROGMEM) - see motion.h
void motionPageInit();

// This function unpacks a motion stored in program memory (Flash)
// in a struct stored in RAM to allow execution
// StartPage - number of the motion page to be unpacked
void unpackMotion(int StartPage);

// This function initiates the execution of a motion step in the given motion page
// Page - number of the motion page
// Returns (long) start time of the step
unsigned long executeMotionStep(int Step);

// This function initializes the joint flexibility values for the motion page
// Returns (int)  0  - all ok
//               -1  - communication error
int setMotionPageJointFlexibility();

// This function executes a single robot motion page defined in motion.h
// StartPage - number of the first motion page in the motion
// Returns StartPage of next motion in sequence (0 - no further motions)
int executeMotion(int StartPage);

// This function executes robot motions consisting of one or more motion
// pages defined in motion.h
// It implements a finite state machine to know what it is doing and what to do next
// Code is meant to be reentrant so it can easily be converted to a task with a RTOS
void executeMotionSequence();

// This function executes the exit page motion for the
// current motion page
 void executeMotionExitPage();

// Function to check for any remaining servo movement
// Returns:  (int)   number of servos still moving
int checkMotionStepFinished();
```

walk.c

```
/*
 * walk.c - Functions for walking
 *
 */

// initialize for walking - assume walk ready pose
void walk_init();

// function to update the walk state
void walk_setWalkState(int command);

// function to retrieve the walk state
// Returns (int) walk state
int walk_getWalkState();

// Function that allows 'seamless' transition between certain walk commands
// Handles transitions between 1. WFWD - WFLS - WFRS and
//                             2. WBWD - WBLS - WBRS
// All other transitions between walk commands have to go via their exit page
// Returns:   (int)  shift flag 0 - nothing happened
//                                                  1 - new motion page set
int walk_shift();

// function to avoid obstacles by turning left until path is clear
// Input:     obstacle flag from last execution
// Returns:   (int) obstacle flag  0 - no obstacle
//                                 1 - new obstacle, start avoiding
//                                 2 - currently avoiding obstacle
//                                -1 - finished avoiding
int walk_avoidObstacle(int obstacle_flag);
```