RIG InMoov Project

Generated by Doxygen 1.8.11

Contents

Chapter 1

Introduction

This is the official documentation for the Conestoga Robotics Innovation Group's InMoov project. It is not meant to be read linearly or all at once. Rather, readers should skim relevant pages and skip information that requires more context. More in depth information is provided for those that wish to modify the source code. Examples are provided when possible for quick reference and experimentation. Note that colored phrases are links for both html and pdf. This project has a git repository located here. You can download the project with the following command after installing git on your system.

```
git clone github.com/rbong/rig-inmoov
```

1.1 Dependencies

The embedd currently depends on Python for an example script, Arduino for the embedded code, Doxygen and texlive (if you wish to remake the documentation), Make (if you wish to build the project) and ROS/LibSerial for the simulation.

1.2 Quick Start

To quickly upload to the right hand, do the following. Copy the **arduino/servo** folder to your Arduino sketch folder. Copy **servo/settings/servo_rhand.h** to **servo/settings.h** in the sketch directory. Copy **arduino/lib/serial.ino** to **servo/serial.ino** in the sketch directory. Open the Arduino software. Select your board and the port it is connected to. Open your sketch, and hit upload. Instructions for the Arduino software are located on its website. Other Arduino files on the bot can be built in a similar way, or the **make** system command can be used.

To transmit commands to the board, run this from the command line.

```
python -i example/servo_demo.py
```

You can now run commands defined in servo_demo.

The python script needs the pyserial library. Its website is located here.

Call this to connect to the board (replace COM0 with the name of the port you set in Arduino).

```
connect ('COM0')
```

To demo some movements, run this.

```
demo ()
```

Note that some movements require input before continuing (press enter).

Read the servo_demo documentation for more. Read the Examples section to understand how it formulates commands.

To run the 3D simulation of the project, please refer to Simulation.

1.3 Installation

You can build this documentation with the command

```
make documentation
```

from the root directory.

To upload the right hand servo code to the connected Arduino, run

```
make servo.ino SETTINGS=servo_rhand ARDUINO_DO=--upload BOARD=nano PORT='/dev/ttyUSB0'
```

Assuming the board is the arduino nano. For the arduino uno, the bord would be uno and the port would be **/dev/ttyACM0**. If multiple boards are connected, or if a board connects and disconnects quickly, the port number may increment. For example, **/dev/ttyUSB1**.

This also requires that the latest version of **Arduino** is installed. Some systems don't have a package for the latest version, such as Ubuntu. Fortunately, the build system creates a valid arduino directory structure. In this case, you can run

```
make servo.ino SETTINGS=servo_rhand
```

And **Arduino** will open. Open a project and navigate to **rig-inmoov/build/servo** and open the .ino file inside.

1.4 Build System

Arduino Directory Structure

The Arduino IDE abstracts many tools for working with Arduino hardware. We choose to use it because of its familiarity and cross system compatibility. However, the Arduino IDE isn't made for command line compilation, and replicating its process requires complex tools not appropriate for this project.

To share code among certain boards, general code is kept in the **arduino/sketch** subdirectories. Inside each subdirectory is an .ino file of the same name, which is the source for the sketch. There is also a **settings** folder in which various headers are kept. These headers are to be moved to **settings.h** inside the sketch folder. For example, code to be shared among servos is kept in **arduino/sketch/servo/servo.ino**, the configuration for the right hand is kept in **arduino/sketch/servo/settings/servo_rhand.h**, and you would move it to **arduino/sketch/servo/settings.h** to use it.

Some sketches require shared code that use Arduino libraries, which means that they need to be contained in .ino files. Arduino includes any .ino files inside a sketch subdirectory. We keep such files in **arduino/lib**. For example, servo boards require serial communication, so we would move **arduino/lib/serial.ino** to the **arduino/sketch/servo** directory.

The reason that shared code among the same board types is treated differently than shared code among different board types is to allow us to keep the settings files short, and also to use them elsewhere in the project that is not necessarily Arduino code. In addition, .ino files do not have a guaranteed order of inclusion.

This is all automated by the build system. If you wish to use the Arduino IDE, either link the appropriate files to your Arduino sketch directory or copy them to the sketch directory and write the changes back later.

1.5 Arduino Protocol

The Arduinos controlling the various parts of the bot communicate over their serial ports. All information is sent and recieved using 8 bit unsigned raw integers instead of text. This allows extremely fast and simple code, but limits programs to 255 signals, identifiers, and values.

However, the Arduino boards only have one function; to pass commands to their servos.

Servo ID

Unless otherwise instructed, the Arduino accepts two values at a time. First, it accepts a servo ID, and a value to write to the pin associated with that servo.

Servo IDs start at 0 and should be uniquely identified on every board, allowing a maximum of 255 servos on each bot, not including signals. They should be assigned different values because if commands are routed through a central board, it allows the protocol to remain the same. If an invalid servo ID is passed to any given board, the board will still read another value from the serial port to avoid syncing issues, but it will do nothing to its servos.

Servo Values

The value passed to the servo can be from 0-180. If it exceeds 180 and is not a valid signal, the board will change the angle to 180. On positional rotation servos, this number represents the servo's target angle. On continuous rotation servos, 90 represents stillness, 180 represents full speed in one direction, and 0 represents full speed in the other.

Each board has callibrations for each servo that scale the angles from 0-180 to some minimum angle and some maximum angle. The host program has no need to know these values. When the board reports current values, they are non-scaled.

To ensure that the board is reading the correct information (servo ID or servo position), there is a cancel signal.

Signals

The boards accept some pre-defined signals that break the default flow. It can recieve these signals at any time and will terminate its current servo command. In addition, the board may print some signals to output. Both kinds of signals, incoming and outgoing, are assigned starting from 255 and going down. In the arduino code, incoming signals are denoted by **∗_SIGNAL**, and the outgoing signals are denoted by **∗_RESPONSE**. We will use this convention.

CANCEL_SIGNAL

This signal exists to cancel all pending input and syncronize the host and master. If a host connects to an Arduino and does not know its current state, it can send this signal and know that the board is waiting on a servo ID.

WAIT_RESPONSE

This response indicates that the board has no bytes left to read and is ready for input. A host does not need to wait for this response, but it may want to if it is experiencing difficulties or the board may have crashed.

START_RESPONSE, END_RESPONSE

These responses mark the beginning and end of a reply to some signal.

DUMP_SIGNAL

This signal makes the board return various information about itself and its servos. As of writing, it returns **START_RESPONSE**, an ID, its number of servos, the IDs of each servo followed by the value last sent to that servo, and **END_RESPONSE**. If a serial connection has access to other boards and is able to send them commands, it may send more servo IDs and servo positions before **END_RESPONSE**. Board IDs start from 181 and go up. It is recommended that if we build more modular bots such as this one in the future using the same protocol, they recieve unique board identifiers. The response is likely to change in the near future as we add sensors and other types of devices. Be sure to regularly check this document.

1.5.1 Examples

```
CANCEL_SIGNAL 0 180 1 135 2 90
```

This would send the servo with ID 0 a value of 180, servo 1 135, and servo 2 90. The board would then respond with **WAIT_RESPONSE**.

```
CANCEL_SIGNAL 0 180 1 135 22 CANCEL_SIGNAL 4 45
```

This would send the servo with ID 0 a value of 180, servo 1 135, begin to read a command for servo 22 but cancel, then send 45 to servo 4.

```
CANCEL_SIGNAL 0 0 1 45 2 90 3 135 4 180 DUMP_SIGNAL
```

On a board with ID **BOARD_ID** and 5 servos identified as 0-4, this would return the response

```
START_RESPONSE BOARD_ID 4 0 0 1 45 2 90 3 135 4 180 END_RESPONSE WAIT_RESPONSE
```

```
CANCEL_SIGNAL 0 0 0 45 0 90 0 135 0 180
```

If this stream of bytes were sent instantly, it would essentially move servo 0 directly to 180 degrees (if it was a positional servo) because of the speed of the commands. However, if we inserted a slight delay, we could slowly move the servo from its starting position to its ending position.

```
CANCEL_SIGNAL 0 0 1 0 0 45 1 45 0 90 1 90 0 135 1 135 0 180 1 180
```

This would move servo 0 and servo 1 from their start to end positions. If we were to send these bytes immediately, they would essentially both move instantaneously to 180 degrees (if they were positional servos). However, if we were to insert a delay, they would both appear to move together slowly to their end location despite the delay between commands. Putting this functionality on the Arduino boards themselves would cause the boards to lock and use up resources, but formulating commands like this allows computation to occur on other systems.

1.5.2 Values

Board IDs

Right hand servo board	181
Right hand flex sensor board	182

Servo IDs

Right hand wrist	0
Right hand thumb	1
Right hand index finger	2
Right hand middle finger	3
Right hand ring finger	4
Right hand pinky finger	5

Signals

CANCEL_SIGNAL	255
WAIT_RESPONSE	254
DUMP_SIGNAL	253
START_RESPONSE	252
END_RESPONSE	251

Chapter 2

Simulation

The simulation for the RIGbot runs in ROS. It accepts commands in the same protocol described in Introduction. You must first use git to download the project, also detailed in Introdcution.

This section only details basic usage and slight modification. For any new features, model additions, or bug fixes, you may need to read the beginner level `ROS tutorials` and the `urdf tutorials`.

2.1 Dependencies

The simulation depends on `ROS`. It is recommended you `install the full system` because of the number of the large number of ROS packages. On Ubuntu, this is achieved with the following commands.

```
sudo sh -c 'echo "deb http://packages.ros.org/ros/ubuntu $(lsb_release -sc) main" >
       /etc/apt/sources.list.d/ros-latest.list'
sudo apt-key adv --keyserver hkp://pool.sks-keyservers.net:80 --recv-key 0xB01FA116
sudo apt-get update
sudo apt-get install ros-jade-desktop-full
```

If you experience problems, please read the full wiki article.

2.2 Building

To build the simulation, simply run the following from the root directory of the project.

```
make simulation
```

You may also want to build manually. Before running any ROS commands, you must run this command.

```
source /opt/ros/jade/setup.bash
```

There are also .zsh and .sh files if you are using different shells. It is recommended you append this command to your shell's .rc file (ex. ∼/.bashrc).

The following only needs to be run once.

```
cd catkin_ws/src
catkin_init_workspace
```

Finally, we can build the project.

```
cd catkin_ws
catkin_make
```

2.3 Usage

Before running any ROS commands relating to the project, you must run the following comand.

```
source catkin_ws/devel/setup.bash
```

There are also .zsh and .sh files if you are using different shells. It is recommended you append this command to your shell's .rc file (ex. ∼/.bashrc). Make sure to fill in the absolute path to rig-inmoov/catkin-ws.

After running all the necessary setup commands, you can finally run the project.

```
roslaunch robot_description display.launch port:=/dev/ttyACM0
```

This requires that an input board such as the flex sensor glove is connected. Replace "port" with the port this board is connected on (search for the naming convention for your board online). The output from launch commands is captured in ∼/.ros/log/.

You can also set and change the port later with the following command.

```
rosparam set /serial_publisher/port /path/to/port
```

The serial publisher does not need to be running.

If the serial publisher crashes because it loses connection, you can re-launch it with this command.

```
rosrun robot_description serial_publisher
```

2.4 Background

We have two custom programs used to maintain the simulation. The first is serial_publisher, compiled from **catkin↩ _ws/src/robot_description/src/serial_publisher.cpp**. This program accepts input from a serial port and translates it to a `ROS topic`.

The other program is state_publisher, compiled from **state_publisher.cpp** in the same directory. This program listens to the topic and translates it into joint positions, which `rviz` (the visualizer component of ROS) listens to. It expects input of the same protocol as described in Introduction.

The modularity of these programs makes it easy to write publishers for new protocols (such as bluetooth) using just **serial_publisher.cpp** as reference. It also makes it easy to add new parts to **state_publisher**. New programs need to be added to **catkin_ws/src/robot_description/CMakeLists.txt**.

The rviz visualizer is loaded with a `urdf` file, stored in **catkin_ws/src/robot_description/model.xml**, that describes how the different 3D models that make up movable parts are positionally related to one another.

2.5 Debugging

There are two example launch files used for debugging, stored in **catkin_ws/src/robot_description/launch** along with **display.launch**. The first is debug.launch, which launches state_publisher and serial_publisher in gdb, and the second, **memdebug.launch**, launches both programs in Valgrind. It is recommended that you use the three launch files as reference when modifying and creating program systems.

Chapter 3

Namespace Documentation

3.1 servo_demo Namespace Reference

Documentation for the servo_demo.py script.

Functions

- def connect (port)

  Sets the global ser variable.
- def sweep (initial, servos, starts, ends, steps)

  Produces complex command chains.
- def unsweep (servos, ends, steps)

  Produces a command chain that resets all servos to 0.
- def servowrite (commands, delay)

  Writes commands to the serial port.
- def gesture (initial, servos, starts, ends, steps, delay)

  Sets the servos to a position, waits for input, then resets them to 0.
- def reset ()

  Immediately sets the servos to 0.
- def cmd ()

  Sends the input to the serial port until a blank line is given.
- def dump (servos=6)

  Retrieves information about the board.
- def peace (delay=0)

  Makes a peace sign.
- def ok (delay=0.01)

  Makes an ok sign.
- def grab (delay=0.02)

  Makes a fist.
- def rockon (delay=0.01)

  Makes a rock on sign.
- def wiggle (n=90, delay=0.01, wiggles=1)

  Wiggles the fingers.
- def count ()

  Counts to 5 on the fingers.

- def demo ()

  Performs all movements.
- def **getarrow** ()
- def calibrate (s=1)

  Calibrate a servo.
- def **calibrate_all** ()

Variables

- int **CANCEL_SIGNAL** = 255
- int **DUMP_SIGNAL** = 253
- int **START_RESPONSE** = 252
- int **END_RESPONSE** = 251
- int RHAND_ID = 181

  The identification byte of the right hand board.
- ser = serial.Serial()

  The serial connection for input and output.

3.1.1 Detailed Description

Documentation for the servo_demo.py script.

An example script for controlling the right hand. This script requires python with a library called pyserial. After instally python 3, pyserial can be installed by calling

```
pip install pyserial
```

Or through your package manager. Example:

```
sudo apt-get install python-pyserial
```

Once you have the library installed, you must have your **PYTHONPATH** variable set to the location of the library.

```
export PYTHONPATH=/usr/lib/python2.7/site-packages
```

Replace 2.7 with your version of python (type in /usr/lib/python and press tab once or twice).

This may vary from system to system. See the `python` and `pyserial` websites for help.

To run the script, execute

```
python -i servo_demo.py
```

or in python

```
import servo_demo
```

This will allow you to call the functions from a command interpreter. It is recommended you experiment with all functions and view the source to understand how you might produce commands. Also try demo() while connected.

The movements included are quickly written and for demo purposes only.

See also

index for the servo codes.

3.1.2 Function Documentation

3.1.2.1 def servo_demo.calibrate (*s* = 1)

Calibrate a servo.

Takes one argument, which is the id of a server. Introduction has the servo codes.

Use the left and right arrow keys to adjust the value of the servo.

Enter the maximum values into the limit array in your settings file.

Definition at line 362 of file servo_demo.py.

3.1.2.2 def servo_demo.cmd ( )

Sends the input to the serial port until a blank line is given.

Must enter integers between 0-255.

Definition at line 218 of file servo_demo.py.

3.1.2.3 def servo_demo.connect (*port*)

Sets the global ser variable.

Note that you must be part of the dialout group in Linux. Enter this command to add yourself to a group.

```
usermod -a -G dialout $USER
```

You must logout to apply group changes. If you cannot logout, you can run the script as root.

```
sudo su
export PYTHONPATH=/usr/lib/python2.7/site-packages
python -i example/servo_demo.py
```

Definition at line 70 of file servo_demo.py.

3.1.2.4 def servo_demo.count ( )

Counts to 5 on the fingers.

See also

sweep()

Definition at line 308 of file servo_demo.py.

3.1.2.5 def servo_demo.demo ( )

Performs all movements.

Definition at line 316 of file servo_demo.py.

3.1.2.6 def servo_demo.dump (*servos* = `6`)

Retrieves information about the board.

The command sends the DUMP_SIGNAL to the serial port and reads back the response. The response is described in servo.ino.

When you write your own similar function, timing is important.

We may need to increase the response delay on the Arduino if you are unable to recieve a response. View the source of this function to understand how it works, but keep in mind it is untested. Also keep in mind that the function is very non-general as it always reads the same amount of bytes instead of looking for the end signal.

Note

While this function could be used to change dynamically between gestures instead of resetting the servos to 0, this function did not exist when this script was first written.

Definition at line 228 of file servo_demo.py.

3.1.2.7 def servo_demo.gesture (*initial*, *servos*, *starts*, *ends*, *steps*, *delay*)

Sets the servos to a position, waits for input, then resets them to 0.

See also

sweep() servowrite()

Definition at line 201 of file servo_demo.py.

3.1.2.8 def servo_demo.grab (*delay* = `0.02`)

Makes a fist.

See also

gesture()

Definition at line 282 of file servo_demo.py.

3.1.2.9 def servo_demo.ok (*delay* = `0.01`)

Makes an ok sign.

See also

gesture()

Definition at line 278 of file servo_demo.py.

3.1.2.10 def servo_demo.peace (*delay* = `0`)

Makes a peace sign.

See also

gesture()

Definition at line 274 of file servo_demo.py.

3.1.2.11 def servo_demo.reset ( )

Immediately sets the servos to 0.

Definition at line 214 of file servo_demo.py.

3.1.2.12 def servo_demo.rockon (*delay* = `0.01`)

Makes a rock on sign.

See also

gesture()

Definition at line 286 of file servo_demo.py.

3.1.2.13 def servo_demo.servowrite (*commands*, *delay*)

Writes commands to the serial port.

If **delay** is 0, immediately writes a list of ints to ser as bytes. Otherwise, writes a list of ints to ser as bytes one at a time.

Inserting a delay controls the speed of the movements.

See also

connect()

Definition at line 184 of file servo_demo.py.

3.1.2.14 def servo_demo.sweep (*initial*, *servos*, *starts*, *ends*, *steps*)

Produces complex command chains.

The output returned from the function is formatted in such a way that the commands can be transmitted to the serial port at variable speeds.

The parameters must all be lists of the same length whose indexes correspond with **servos**.

Parameters

initial	An array of initial commands to append to.
servos	The servos to control.
starts	The current positions of the servos.
ends	The desired end positions.
steps	The values to use to decrement/increment. Converted to absolute value.

Returns

Upon success, returns the generated list of commands. The commands are surrounded by the CANCEL_↩ SIGNAL for syncing. The servos will always reach their exact final destination, unless if **steps** at the index of the servo is 0. Upon error, raises **'Invalid sweep command'**. This indicates non-matching list lengths.

Example

```
sweep ([], [1,2], [180,0], [90,180], [90,45])
```

Returns

```
[CANCEL_SIGNAL, 1, 180, 2, 0, 1, 90, 2, 45, 2, 90, 2, 135, 2, 180, CANCEL_SIGNAL]
```

The protocol this follows is described in servo.ino. By intertwining commands in this way, a delay can be inserted between bytes and fluid motion is still preserved.

Definition at line 92 of file servo_demo.py.

3.1.2.15 def servo_demo.unsweep (*servos, ends, steps*)

Produces a command chain that resets all servos to 0.

See also

sweep()

Definition at line 176 of file servo_demo.py.

3.1.2.16 def servo_demo.wiggle (*n* = `90`, *delay* = `0.01`, *wiggles* = `1`)

Wiggles the fingers.

See also

sweep()

Definition at line 290 of file servo_demo.py.

3.1.3 Variable Documentation

3.1.3.1 servo_demo.ser = serial.Serial()

The serial connection for input and output.

Must be initialized.

See also

connect()

Definition at line 68 of file servo_demo.py.

Chapter 4

File Documentation

4.1 arduino/lib/serial.ino File Reference

```
#include <SoftwareSerial.h>
```

Functions

- SoftwareSerial **cmd_serial** (CMD_RX_PIN, CMD_TX_PIN)
- void **serialSetup** ()
- bool **serialCmdAvailable** ()
- void **serialCmdWait** ()
- void **serialCmdWrite** (uint8_t c)
- void **serialDebugWrite** (uint8_t c)
- uint8_t **serialCmdRead** ()
- int **serialCmdGetByte** ()
- uint8_t **serialDebugRead** ()
- int **serialDebugAvailable** ()
- void **serialDebugPrint** (const char ∗s)
- void **serialDebugPrintInt** (int i)
- void **serialDebugPrintIntPretty** (const char ∗pre, int i, const char ∗post)

4.1.1 Detailed Description

This is the serial communication shared code.

Serial communication is broken up into two serial ports; a command port, and a debug port. On boards that only recieve commands (servo controllers), the command port is the port on which the board recieves commands and prints responses, and the debug port is the port on which it prints information if the VERBOSE constant is true.

On boards that control other boards (flex sensors), the command port is the port on which it transmits commands to and listens for responses from the child board, and the debug port is the port on which it recieves any commands and prints information.

To use the USB serial port on an Arduino board, simply set the board's corresponding RX and TX pins for output (usually 0 and 1).

4.2 arduino/sketch/flex/flex.ino File Reference

```
#include "settings.h"
```

Enumerations

- enum {
  CANCEL_SIGNAL = 255, **MIN_SIGNAL** = 250, **MAX_SIGNAL** = 249, **START_RESPONSE** = 252,
  END_RESPONSE = 251, **CALIB_RESPONSE_LEN** = FLEXS ∗ 2, **MIN_LIM** = 0, **MAX_LIM** = 1 }

Functions

- void **setup** ()
- void **loop** ()
- int **analogGetInt** (uint8_t flex_index)
- int **getAdjustedFlex** (uint8_t flex_index, int flex_amount)
- int **getFlexMax** (uint8_t flex_index)
- int **getFlexMin** (uint8_t flex_index)
- void **softSerialServoCmd** (uint8_t servo_id, int servo_angle)

Variables

- int **current_pos** [6] = { 0 }

4.2.1 Detailed Description

This is the code for the flex sensor board.

It follows the same settings convention as serial.ino. It uses the serial.ino shared code to transmit commands to a servo board given a settings file. Programs generated with this file translate values read from variable resistance (flex sensors) hooked up to voltage divider into values from 0-180 for the servo controller to read.

4.3 arduino/sketch/flex_double/flex_double.ino File Reference

```
#include "settings.h"
```

Enumerations

- enum {
  CANCEL_SIGNAL = 255, **MIN_SIGNAL** = 250, **MAX_SIGNAL** = 249, **START_RESPONSE** = 252,
  END_RESPONSE = 251, **CALIB_RESPONSE_LEN** = FLEXS ∗ 2, **MIN_LIM** = 0, **MAX_LIM** = 1 }

Functions

- void **setup** ()
- void **loop** ()
- int **analogGetInt** (uint8_t flex_index)
- int **getAdjustedFlex** (uint8_t flex_index, int flex_amount)
- int **getFlexMax** (uint8_t flex_index)
- int **getFlexMin** (uint8_t flex_index)
- void **softSerialServoCmd** (uint8_t servo_id, int servo_angle)
- void **softSerialServoDebug** (uint8_t servo_id, int servo_angle)

Variables

- int **current_pos** [6] = { 0 }

4.3.1 Detailed Description

This is the code for the flex sensor board.

It follows the same settings convention as serial.ino. It uses the serial.ino shared code to transmit commands to a servo board given a settings file. Programs generated with this file translate values read from variable resistance (flex sensors) hooked up to voltage divider into values from 0-180 for the servo controller to read.

4.4 arduino/sketch/servo/servo.ino File Reference

```
#include <Servo.h>
#include "settings.h"
#include <stdint.h>
```

Enumerations

- enum {
  MIN_LIM = 0, MAX_LIM = 1, CANCEL_SIGNAL = 255, WAIT_RESPONSE = 254,
  DUMP_SIGNAL = 253, START_RESPONSE = 252, END_RESPONSE = 251, DUMP_RESPONSE_LEN = SERVOS ∗ 2 + 4 }

  The print buffer.

Functions

- void setup ()
- void loop ()
- void serialPrint (const char ∗s)
- void serialPrintInt (uint8_t i)
- void serialPrintIntPretty (const char ∗pre, uint8_t i, const char ∗post)
- int serialGetByte ()
- void setServoFromIndex (uint8_t servo_index, uint8_t servo_angle)
- void setServoFromID (int servo_id, uint8_t servo_angle)
- void dump ()
- void serialWait ()

Variables

- Servo servo [SERVOS]
- uint8_t current_pos [SERVOS] = { 0 }
- char **buf** [BUFSIZE]

4.4.1 Detailed Description

This is the common code for all Arduino servo boards.

The program is written to be a simple module that can pass values to servos. Each board needs its own settings file. Because of limitations of the Arduino command line interface, these are included as **settings.h**. The Makefile manages which configuration is currently located in the same directory as this file and named **settings.h**. A better build system in the future could use the tools **avrdude** or **ino**. Other Arduino boards in the project, such as flex sensor boards, use an identical folder structure and include process.

See rhand.h for an example of a settings header, and Arduino Protocol for information servo IDs and other relevant values.

4.4.2 Enumeration Type Documentation

4.4.2.1 anonymous enum

The print buffer.

Enumerator

MAX_LIM Index for the limit variables.

CANCEL_SIGNAL Index for the limit variables.

WAIT_RESPONSE Incoming signal commanding termination of the current loop.

DUMP_SIGNAL Outgoing signal indicating that we are waiting for input.

START_RESPONSE Incoming signal indicating that we are to print information to serial.

END_RESPONSE Beginning of some multi-byte response.

DUMP_RESPONSE_LEN End of some response.

Definition at line 43 of file servo.ino.

4.4.3 Function Documentation

4.4.3.1 void dump ( )

Writes various information about the board to the serial port. Triggered on reception of DUMP_RESPONSE_LEN. Transmits the ID of the board, the number of servos, each servo ID, and the current position of the servos.

See also

Arduino Protocol, START_RESPONSE, END_RESPONSE, current_pos, getServoIDFromIndex()

Definition at line 263 of file servo.ino.

4.4.3.2 void loop ( )

The **loop** function is called continually until the program exits. It performs actions based on Arduino Protocol. Recieves two unsigned 8-bit integers, a servo id and a servo angle, then calls setServoFromID(). If the DUMP_SI↩GNAL is recieved at any time, calls dump() and continues. If the CANCEL_SIGNAL is recieved, it does nothing and continues. If, at the beginning of the function, there is no pending input, it transmits WAIT_RESPONSE, then does nothing until input is available.

Definition at line 98 of file servo.ino.

4.4.3.3 int serialGetByte ( )

Retrieves a byte from the serial port.

Returns

If a normal value is recieved, returns that value as an integer. If CANCEL_SIGNAL is recieved, returns -1.

Definition at line 178 of file servo.ino.

4.4.3.4 void serialPrint (const char ∗ *s*)

Prints a string to the serial port if VERBOSE is enabled.

Definition at line 140 of file servo.ino.

4.4.3.5 void serialPrintInt (uint8_t *i*)

Prints an integer to the serial port as a string if VERBOSE is enabled.

Definition at line 151 of file servo.ino.

4.4.3.6 void serialPrintIntPretty (const char ∗ *pre*, uint8_t *i*, const char ∗ *post*)

Prints an integer as a string surrounded by two strings to the serial port if VERBOSE is enabled.

Definition at line 163 of file servo.ino.

4.4.3.7 void serialWait ( )

Waits for serial input to become available then returns.

Definition at line 289 of file servo.ino.

4.4.3.8 void setServoFromID (int *servo_id*, uint8_t *servo_angle*)

Writes an angle to a pin associated with a servo ID.

See also

setServoFromIndex(), getServoIndexFromID()

Definition at line 242 of file servo.ino.

4.4.3.9 void setServoFromIndex (uint8_t *servo_index*, uint8_t *servo_angle*)

Writes an angle to a pin associated with a servo index. Servos are controlled by sending varying pulse widths over their signal wire. The adjusted angle will be sent to the servo. Updates current_pos with the non-adjusted angle. If the servo index is invalid, the function does nothing and exits, but if the angle is out of range, it is changed to a valid value.

Parameters

servo_index	The servo to write to.
servo_angle	The angle to write after adjusting. It may be reversed.

See also

reverse

Definition at line 203 of file servo.ino.

4.4.3.10 void setup ( )

The **setup** function is called at the beginning of the program. In it, we assign a pin to each servo.

Definition at line 59 of file servo.ino.

4.4.4 Variable Documentation

4.4.4.1 uint8_t current_pos[SERVOS] = { 0 }

The current position of each servo. These are updated whenever a servos value is changed and retrieved upon DUMP_SIGNAL.

Definition at line 39 of file servo.ino.

4.4.4.2 Servo servo[SERVOS]

The representation of the servos. Used to keep track of pin assignments and send output.

See also

getServoIndexFromID(), Arduino

Definition at line 33 of file servo.ino.

4.5 arduino/sketch/servo/settings/servo_rhand.h File Reference

Macros

- #define SERVOS 6
- #define BUFSIZE 32

Enumerations

- enum {
  VERBOSE = 0, BOARD_ID = 181, **DEBUG_RX_PIN** = 1, **DEBUG_TX_PIN** = 0,
  CMD_RX_PIN = 1, **CMD_TX_PIN** = 0, **SERIAL_BAUDRATE** = 9600 }

Functions

- int getServoIndexFromID (uint8_t servo_id)
- int getServoIDFromIndex (uint8_t servo_index)
- int getServoPinFromIndex (uint8_t servo_index)

Variables

- uint8_t limit [SERVOS][2]
- uint8_t default_pos [SERVOS] = { 0, 0, 0, 0, 0, 0 }
- uint8_t reverse [SERVOS] = { 1, 1, 1, 1, 1, 1 }
- uint8_t pin_offset = 2

4.5.1 Detailed Description

This is an example **settings.h** for servo.ino.

Every Arduino board on the bot is to be a simple servo controller that accepts commands from serial input, and so they are all to share the code located in servo.ino. Each board needs its own header like this one to define custom information about the board and each servo on the board. This is such a settings file and will be documented as an example.

4.5.2 Macro Definition Documentation

4.5.2.1 #define BUFSIZE 32

Size of the print buffer (must be large enough to hold DUMP_RESPONSE_LEN).

Definition at line 18 of file servo_rhand.h.

4.5.2.2 #define SERVOS 6

Number of servos to assign.

Definition at line 14 of file servo_rhand.h.

4.5.3 Enumeration Type Documentation

4.5.3.1 anonymous enum

Enumerator

VERBOSE	Whether to print debug output to the serial port.
BOARD_ID	The unique identification for this board.

Definition at line 54 of file servo_rhand.h.

4.5.4 Function Documentation

4.5.4.1 int getServoIDFromIndex (uint8_t *servo_index*)

This function gets servo ID using its array index. If the servo index is out of bounds, it returns -1. This must correspond with getServoIndexFromID().

Definition at line 87 of file servo_rhand.h.

4.5.4.2 int getServoIndexFromID (uint8_t *servo_id*)

This function gets array index of a servo using its ID. If the servo does not belong to this board, it returns -1. It is recommended that boards with complex ID assignments use a switch statement or a similar solution to save resources. This function must correspond with getServoIDFromIndex().

See also

Values

Definition at line 74 of file servo_rhand.h.

4.5.4.3 int getServoPinFromIndex (uint8_t *servo_index*)

This function returns the desired pin of a servo given its index number, allows complex pin assignment. If the servo idex is not valid, returns -1.

Definition at line 100 of file servo_rhand.h.

4.5.5 Variable Documentation

4.5.5.1 uint8_t default_pos[SERVOS] = { 0, 0, 0, 0, 0, 0 }

The default position to set each servo to. Its index corresponds with servo.

Definition at line 42 of file servo_rhand.h.

4.5.5.2 uint8_t limit[SERVOS][2]

Initial value:

```
= {
    { 125, 160 }, {  50, 160 }, {  55, 155 },
    {  35, 155 }, {  45, 145 }, {  55, 145 },
}
```

This is the callibration for each servo- its minimum then maximum angle (for positional servos) or its minimum then maximum speed (for continuous rotation servos). Its index corresponds with servo.

Definition at line 34 of file servo_rhand.h.

4.5.5.3 uint8_t pin_offset = 2

Pin value to begin assigning pins at. Unique to this settings file.

See also

getServoPinFromIndex()

Definition at line 52 of file servo_rhand.h.

4.5.5.4 uint8_t reverse[SERVOS] = { 1, 1, 1, 1, 1, 1 }

Indicates whether to reverse the angles for each servo, for servos that turn the wrong way. Its index corresponds with servo.

Definition at line 47 of file servo_rhand.h.

Index