Servo Documentation

This is a first draft of the OpenServo documentation. This document will detail how to build, program and interface to the OpenServo. Drafted by Barry Carter.

Any references to register names can be resolved to register address using this table

Sections:

1. Terminology
2. Construction of the OpenServo
3. Bootstrapping the OpenServo
4. Programming the OpenServo
5. Hardware interface to the OpenServo

6. Interfacing with the OpenServo using TWI (I2C)

7. Example Communication strings
8. Debugging and Caveats

1) Terminology

The OpenServo project combines many resources to create the final servo controller board. There is the hardware side, which consists of an MCU host controller, h-bridge driver and power supply and management. There is also a software component to the OpenServo. The software consists of three main parts. There is the embedded code on the MCU, which is two of the three parts, and the host controller software.

OpenServo - Open Source in motion. OpenServo is a small board that is designed to fit inside of popular servo hardware. The OpenServo board is designed to replace the existing hardware in order to provide more functionality. The OpenServo is an advanced platform that enables multiple servos to be controlled with very high precision, accuracy, and repeatability. The OpenServo project consists of two main Open Source modules. At the current time, OpenServo has an Open Source hardware design, and an Open Source software portion.

Hardware components

The Hardware can be broken down into four sub components, each of which will be described in further detail.

- MCU
- H-bridge
- Power supply
- Measurement

MCU This is the main controller for all of the OpenServo functions. It serves as the I2C slave controller, as well as coordinating all of the functions of the servo. This is programmed from the PC host using avr-gcc tools.

H-Bridge The H-Bridge is designed to switch high current to the motor within the servo. There are two input lines that are driven by the MCU that control the speed and direction of the servo motor.

Power Supply The OpenServo has a small +5v regulator on board to isolate any possible noise from the h-bridge from interfering with the MCU.

Recommended values for the power supply are:

At least 6v. Max 8-9v

Current at least 500ma *_per servo_*

Measurement The OpenServo is constantly measuring it's external environment to check that the servo is within it's programmed parameter limits. The MCU measures the voltage across the potentiometer in the servo to check that it is always in the correct position. Additionally, the OpenServo measures the current through the servo, and also the battery voltage levels.

Software Components

The OpenServo consists of three main software components.

- Embedded Bootloader
- Embedded Application code
- Host Control software

Embedded bootloader When the OpenServo is powered on, the first thing that is executed is the bootloader. The bootloader allows for new application code to uploaded within 3 seconds of power on. The bootloader initialises at the standard address of 0x7F by default. This behaviour can be altered.

Embedded Application code Once the bootloader has timed out, the application code is booted. The application code is the main controller of the servo hardware. The main application code is uploaded to the servo from the bootloader, facilitating simple updates of the OpenServo application firmware.

Host Control software Once the OpenServo is booted and running, you can communicate with it using the TWI (I2C) interface. You will need a host controller to communicate with the OpenServo, whether that is a PC to I2C bridge, or another MCU.

2) Construction of the OpenServo

If you intend to construct your own OpenServo, it is advised you consult the forums before you begin. The forums are the place to learn about new features, and also any pit falls you might be expected to come across.

3) Bootstrapping the OpenServo

This is the first stage of getting the program code onto the OpenServo.

The preferred development platform for the OpenServo is the Atmel STK500 development board. This has headers to program the OpenServo bootloader. The

complete guide to flashing an OpenServo using the STK500 is here http://www.OpenServo.com/moin.cgi/ServoBootstrapping

Although the STK500 is recommended for development of the OpenServo, it is not necessary to use one to flash the bootloader stage. You can purchase a relatively cheap Kanda STK200 programmer, or alternatively build your own. Resources for using both the STK200, and the homebrew are detailed here. http://www.OpenServo.com/moin.cgi/AvrDudeBootstrapping

4) Programming the OpenServo

Once the bootloader firmware is flashed onto the OpenServo board, you can upload the main application code.

The bootloader itself doesn't control the servo, but instead provides the bare essential TWI interface in order to upload the new application code. The bootloader allows unprotected read/write access to the main application code area, for easy I2C flashing.

Once the servo is in the bootloader, you have three seconds to address the device at it's default bootloader address of 0x7F. Once this times out, the bootloader code will try to execute the main application.

You can use the Windows application, and a Dimax I2C bridge to program your servo. The application details are available here http://www.OpenServo.com/moin.cgi/ServoProgramming

If you don't have access to a Dimax I2C bridge, you can write your own flash routines, or use the API library to handle the functions. It is fairly simple to flash the servo, so if you have a basic understanding of the I2C communication method with your preferred hardware, you can make your own flash routine.

Addresses in the bootloader are 16 bit long, and start at 0x0000. The bootloader reserves the lower portion of flash, so any application code must be compiled with this offset taken into consideration.

It is advisable when flashing to send the data in small pages, rather than one large page of data. It is recommended that the pages are sent in 64 byte blocks, and then verified for data corruption.

Once you have uploaded the new application code, you can send the "Magic address" of 0xFFFF to reboot into the application code, or just cycle the servo power.

5) Hardware Interface to OpenServo

- Dimax example
- Gumstix example
- MCU host example

6) Software Interface to OpenServo

Communicating with the OpenServo is very simple. It uses basic TWI commands to set it's position, and read variables from the servo.

- I2C resources.
- Software API

Setting the servo position

A standard I2C transaction at the byte level is simple. First you send the device slave address, and then you send the register to read/write to. After that is sent you can either write some data to the register(s) or read data from those registers. Address reading and writing is incremental, and as such when you read, or write more than one byte to any given start address, it will automatically increment the pointer position.

For example, to set a servo position of 980 (which is the max value I can get from a Stell Servo) you would send this command sequence.

Assuming device address = 0x20

I2C communicates using hex bytes, so position 980 translates to 0x03D4

As with any I2C transaction, this 16 bit value must be split into 2 8 bit bytes. All that is required, is to send the upper and lower nibbles of the byte.

send: 0x20 0x10 0x03 0xD4

Setting the position and speed

When writing a speed to the servo, you don't use the standard seek register. Instead of writing to SEEK_HI/LO (0x10/0x11), you must write to the SEEK_NEXT_HI/LO (0x12/0x13).

The servo will not perform the movement until a valid speed variable is set with either of the two speed algorithms available.

There are two methods of speed control of a servo, Accumulation based speed and Time based speed.

Speed based works this way.

Accumulation based speed

From the forums by Mike Thompson:

Setting speed to 0x0100 (1.0) will yield a speed of 100 units a second - a fairly slow speed. The servo I'm testing with (Futaba S3003 hardware) has a maximum speed of about 16 units in one ADC sample (1/100th of a second). Setting the speed more than 0x1000 (16.00) doesn't make the servo move any faster, but other faster servo hardware could conceivably support higher speeds such as 0x2000 (32.00) or more. Using slow speeds such as 0x0080 (0.5) results in a nice slow slew of the servo to the new position.

Timed movement

Time based movement works by setting a time to make the servo move. A value of 100 represents a time span of one second.

Example:

To set the speed to 100 units per second, with a position of 980, send this command sequence:-

definitions:

0x20 = device address

0x12 = SEEK_NEXT_HI

0x14 = SEEK_SPEED_HI

send position:

send: 0x20 0x12 0x03 0xD4

send speed:

send: 0x20 0x14 0x01 0x00

In the first step we send the new position, and in the second transaction, we send the speed

Reading useful variable from the servo

The servo contains a goldmine of useful information. You can read any of the variables from the servo even while it is motion. This way you can monitor the

servo and it's environment to achieve the best from you application. You can get information such as servo position, speed, and current draw as well as the voltage of a battery.

Potential applications using servo variables include:-

- Sensing current to detect objects. This has been demonstrated on a hexapod leg to check for uneven ground.
- Constant speed control. Although this functionality is not currently implemented in the OpenServo, you can make the servo sweep with a constant speed.
- Battery sensing. You could read the current battery voltage variable, and control a switching mechanism to control battery loads.
- Position

You can read the current position from the servo by reading from POSITION_HI/LO.

- Velocity

Velocity registers store the current servo velocity. This variable is VELOCITY_HI/LO

- Current

The amount of current that is being drawn through the H-bridge can be read from address POWER_HI/LO

All variables are available to read, please consult the register address table for more information.

Configuring your servo

You can configure many useful variables on the OpenServo to match your particular hardware and setup. These are explained below.

In order to write any configuration variables to the servo, you must first enable the write to the registers. Sending WRITE_ENABLE accomplishes this. The values that you change will not be persistant over a servo reboot unless you save the registers to the internal eeprom.

- PID/IPD

OpenServo uses advanced control loops to make movement configurable, and

smooth. These control loops make sure the motor is started slowly, and then incrementally ramped up to it's desired speed. The same is true for the motor coming to a stop, but in reverse.

OpenServo currently has three implementation for the controll loop that is used. PID, IPD and a state estimator. PID motion is the default, and only this will be described here

You may have to tune your servo PID values for each application. Although the default values are suitable for the most basic use, you will get the best from your servo if the values are tweaked to gain optimum settle and ramp times for the servo.

You can configure the PID values for the OpenServo by setting these variables.

PID_PGAIN_HI/LO

PID_DGAIN_HI/LO

PID_IGAIN_HI/LO

- Limits

You can configure the limits that the OpenServo can effectively use by setting the MIN_SEEK_HI/LO and MAX_SEEK_HI/LO register values. This will protect the servo mechanism from damage if a value written to the servo is out of the physical hardware's scope.

- Deadband

The deadband is defined as the area of the servo range where no action is applied to the motor. The default value for the deadband is 2 units.

You may need to apply the deadband variable if your servo horn oscillates around it's destined position. This is usually caused by badly tuned PID values, or an abnormal load that causes lots of stress on servo.

The width of the deadband is configured using the DEADBAND variable, which is an 8 bit value.

- TWI/I2C Address

The TWI_ADDRESS variable lets you change the I2C address of the servo from it's default. You will need to put each servo on the bus on it's own address. If any two devices share an address, there will be unreliable communications, and possible servo damage.

Sending servo commands

Commands can be sent to the OpenServo that perform certain actions. These commands require no data after the register select byte.

A typical command write sequence look like this:

send: 0x20 0x80

- Reset

Sending the command RESET will immediately reboot the servo.

- Enable/disable PWM

You can enable or disable output to the motor using the commands WRITE_ENABLE and WRITE_DISABLE

- Save/unprotect

In order to set new configuration values to the OpenServo, you must first unprotect the registers.

Sending the command WRITE_ENABLE will enable write of read/write protected registers.

Likewise, sending WRITE_DISABLE will disable write of read/write protected registers.

To make the configuration values persistant across reboots, send REGISTERS_SAVE

If you have changed some configuration variables, and wish to revert to the saved values, send REGISTERS_RESTORE

In order to restore the OpenServo register values to initial default, send REGISTERS_DEFAULT

Example Communication

TBD

Debugging and Caveats

Debugging

If for any reason you have any problems with the OpenServo platform, there are a few things you can look out for to help.

Problem:

No Communication with the OpenServo despite being able to communicate with my XYZ I2C device. Help!

Solutions:

This is most probably a hardware issue. If you have no I2C communications at all, check to see if the device is on the bus at a different address. It has been known for the servo to stay in bootloader mode if there are hardware issues. This is caused by a short circuit continually rebooting the servo once it initialises the main application code. You should check for shorts between any part of the H-Bridge of the circuit. if this is the case, you will see the device at 0x7F.

If you still cannot see the servo on the bus, check all connections to the servo. make sure it has adequate power to run. The servo needs at least 6v to perform within it's designed limits. You can safely run the servo from any voltage between the range of 6-8v. Voltages outside of this range are untested. Check the circuit for solder bridges, and any misplaces components.

Problem:

The servo reboots while it is moving. This is intermittent.

Solutions:

This is most likely caused by a poor power supply. You need to make sure that the servo runs from at least 6v. If you have problems at these levels, try raising the voltage a little. You should also use a high current PSU. If the current is too low, the motor will saturate the supply, and the fall in voltage reboot the MCU.

If you are confident that the power supply is working, and is supplying enough current, you can check to make sure you don't have any fine whiskers of solder on the board.

Problem:

I keep getting corrupt data when writing to the servo.

Solutions:

I2C bus noise can interfere with the data that is flowing. Try to isolate the I2C bus cables from the power cables, and any other source of noise.

If you still have problems with bus noise, it may be a termination problem. If the capacitance between the I2C lines falls out of specification, communications can become unreliable. Consider using a LTC1694 I2C bus terminator.

You must also check to make sure the I2C bus has adequate pullup resistors. A general value is around 2K

Problem:

My I2C bus host is using 3.3V while the OpenServo is 5v. How can I convert between these. I tried resistors in a potential divider, but it just isn't cutting the mustard.

Solutions:

Well, you can use a bus level shifter. This will safely convert the voltages between the two, and is very simple to build your own. If you want to use a predesigned PCB, you can download the image masks from this robotics site

Caveats

If you have more than one OpenServo on the I2C bus, and you power on simultaneously, you must wait for each servo to timeout from the bootloader before you initialise communication. This is an issue with all the servos booting with the bootload I2C address of 0x7F. If you attempt to communicate with a servo at 0x7F, you may cause serious servo confusion, leading to hardware damage.

ServoDocumentation (last edited 2012-10-12 19:51:18 by localhost)

- Immutable Page
- Info
- Attachments
- More Actions: Raw Text Do

- MoinMoin Powered
- Python Powered
- GPL licensed
- Valid HTML 4.01