A SUN LABS RESEARCH PROJECT

Project Sun SPOT

Remote Control Camera

How to attach Servo Motors to a Sun SPOT to create a remote-controlled pan-and-tilt Web Camera including BaseStation webserver for web-based control.

While pan-and-tilt camera mounts are plentiful and cheap these days, the ability to quickly and easily create your own pan-and-tilt camera mount based on SUN SPOTs is still an interesting and worthwhile project to pursue as an introductory exercise in how to develop full-featured Sun SPOT applications.

As part of this development exercise, I have also developed some extensions to the BaseStation itself which I use to dynamically add, and update, BaseStation applications, but those additions will be detailed in a separate Tech Note rather than being included here.

Connecting Sensors and Actuators, Development and Debugging on the Sun SPOT Platform

Part of the design goals of the entire Sun SPOT Project was to make connecting and controlling external devices – both sensors and actuators – as easy and as straightforward a process as possible. This project was an exercise in validating that goal, and testing to see how easy the process was, and what, if any, difficulties or issues might be encountered by an end user in connecting servos to the Sun SPOT platform. Once connected, control of the servos through software needed to be thoroughly tested, and finally, in order to add some real-world usefulness to the project, the ability to control the SPOT and its associated servos remotely, specifically via a web-interface, was developed.

Completing this entire project, from wiring the servos to developing the control software to integrating the entire thing with a web service took about a day of diligent work by a single developer. All development, debugging and testing was done using the NetBeans Integrated Development Environment (IDE) including using the remote deployment and debugging capabilities of the Sun SPOT platform.

What you'll need to complete this project

I will provide a short materials list here, but feel free to substitute other, more convenient or available parts as you see fit. Remember, this was my project, and yours can, and probably should, be slightly different based on your needs, your skill-level, and your available components.

1. 2 Servos. I used 2 HiTech servos from an RC Car supply store. Specifically an HS-311 and an HS-81MG, but any servos should suffice.
2. 2 Sun SPOTs. One base station and one with a Sensor board for connecting the servos
3. 1 120V AC to 5.0v DC adapter (I used an old one I had laying around)
4. 1 Firewire camera (I used an Apple iSight, but you could use any USB or Firewire camera)
5. Standoffs, crimp-style wire connectors, misc. screws, nuts and wire ties
6. Mounting block for the Servos/Camera

You will also probably need a soldering iron in order to connect the servos to the Sun SPOT. A USB Wall Charging unit to power the Sun SPOT might also be useful, though this is not specifically required in order to make the project complete.

Wiring the Servos and Sun SPOT

The wiring of the Servos and the Sun SPOTs was very easy and straight forward. The Servos involved(made by HiTec) require more power than the High Current pins can reliably deliver, so an external power source was needed to power the servos. Given that my lab has a plethora of leftover wall-warts from various electronic devices, I just grabbed one that seemed to produce the correct amount of power for the servos and wired it directly to the servos themselves. All ground wires were tied together to the Sun SPOT ground pin to provide a common ground.

The servo control wires were each hooked to a High Current pin in this example, though use of the High Current pins was not required. Each servo was connected to a separate pin (tilt was connected to pin H1, and pan was connected to pin H2, though the exact pin numberings could be changed without any issues as long as pin changes were also made in the software. For a complete schematic of how the entire project was wired together, please refer to Figures 1 and 2. Figure 2 shows the details of the header on the eDemoSensorBoard and which wires were soldered to which positions.

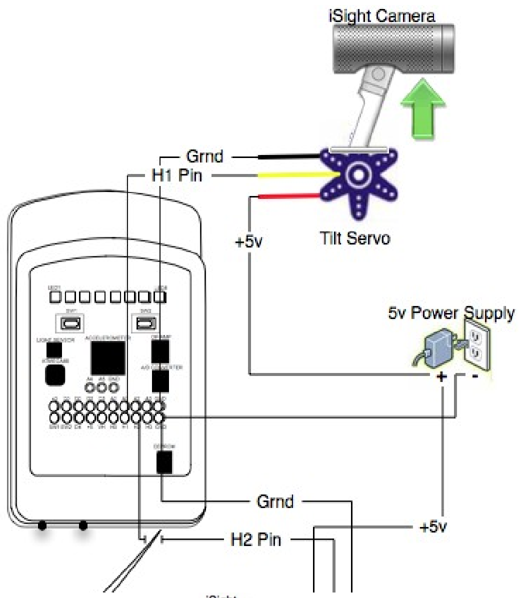
All of this wiring required very little soldering to complete. The recycled wall power supply had to be slightly modified by removing the existing barrel-plug and connecting the positive and negative wires to the positive terminals of the servos and the ground pin of the Sun SPOT (I used a simple volt meter to determine which wire was positive and which was negative once I had removed the barrel connector).

Once that wiring was completed, the control-wires of the servos were connected directly to the Output Control pins on the Sun SPOT.

Then all ground wires (from the 2 servos and from the wall power supply) were connected together and tied to one of the ground pins on the eDemoSensorBoard.

This completed all of the required wiring for the project. With the exception of the connections to the eDemoSensorBoard pins, the other connections could all be made without soldering at all. Connecting to the eDemoSensorBoard, however, does require soldering, though the soldering skills required are minimal.

To avoid any issues of battery life on the Servo-control Sun SPOT a USB Power Adapter was used to provide continuous power to the Sun SPOT. This power adapter required no



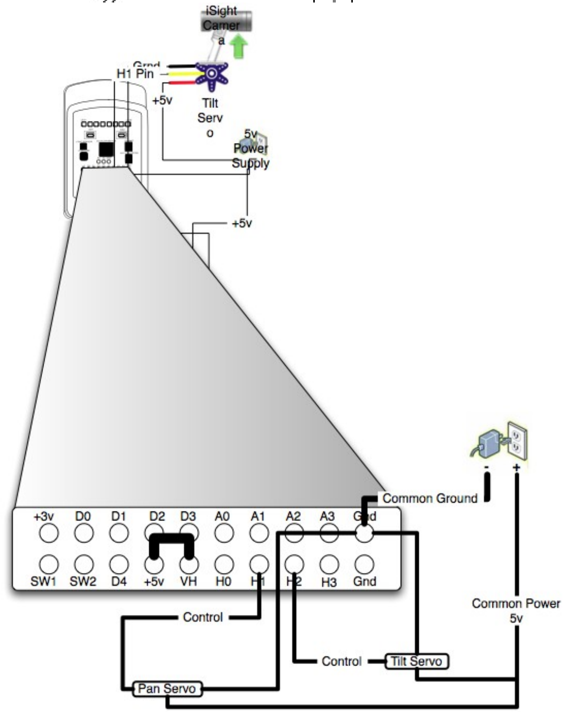


Figure 2: Pin Header Detail

additional wiring as a standard USB mini-B power adapter provides the required connector and the correct voltage for continuous operation of a Sun SPOT.

Sun SPOT Servo Control Software

While the actual code to control a Servo from a Sun SPOT is extremely easy to write, I was interested in expanding on basic servo control and testing some of the performance capabilities of the Sun SPOTs themselves. In order to push the Sun SPOTs a bit further, I made each servo controller run in a separate Thread inside the VM, and each Servo Controller also listened for control commands on a Message Queue. By moving control of each servo to its own thread I allow for the ability to move the camera in 2 directions simultaneously rather than having to wait for each servo operation to complete before beginning the next servo operation.

The Servo Control project consist of only two classes: `CameraController.java` and `ServoDrive.java`. The `ServoDrive` class is a more generic Servo Controller class that handles the direct manipulation of the Servos . The `ServoDrive` class implements Runnable so that it can run in its own Thread, thereby enabling the simultaneous operation of several servos. This class could be re-used to control any number of servos. Some of the other important aspects of the `ServoDrive` class include:

- Name: The ability to name the Servo
- Min, Max and Center points defined: Each `ServoDrive` object allows for the definition of the Minimum value (the bottom-end of the servo's range), the Maximum value (the top-end of the servo's range) and the center point of the servo's range.
- Center: Causes the servo to automatically move to its defined centerpoint.
- Scan: Causes the servo to move through its entire range of motion and return to the current resting point.
- Move: the motion of the servo itself (even within the `ServoDrive` class itself) is all done through a `MoveServo` handler which smooths the motion of the servo and prevents the servo from moving too rapidly which would cause blurring of the image. The speed of the servo's motion can be controlled by setting a sleep property of the `ServoDrive` object (and in fact this value is over-ridden during the scan operation mentioned above to provide for a faster, smoother scan of the servo's range).

ServoDrive Class Object

The `ServoDrive` class object is instantiated by the following Object Creation Method:

```
            public ServoDrive(Servo drive){
                    this.myServo = drive;
                }
```

Of course, this implies that the Servo itself, and its control-pin, is claimed by the instantiating object and indeed the controlling Object should have already claimed the required Pin as a Servo pin.

While not initialized during object creating, the `ServoDrive` message queue must be initialized before starting the thread or the operation will fail. I chose not to include an Object Creation Method which would also initialize the Queues (since the Object takes both an input and an output queue) but instead rely on get and set methods for initializing the queues.

Once the `ServoDrive` Object has been properly initialized, a new Thread object can be created and started which will cause the `ServoDrive` Object to center the controlled servo (based upon its center-point) and then begin listening for movement messages.

Moving the servos is all done in small increments to make the servo movement smooth and even while keeping it from being too rapid and 'jerky.' To accomplish this, I use the `moveServo()` method in the `ServoDrive` class, illustrated here:

```
    private void moveServo(int amt, char dir){
            if(dir == '+'){
                if(myServo.getValue() + amt < servo_max){
                    int tgt = myServo.getValue()+amt;
                    for(int x = (myServo.getValue() + 10); x <= tgt; x+=10){
                        myServo.setValue(x);
                        try {
                            Thread.sleep(servo_sleep);
                        } catch (InterruptedException ex) {}
                    }
                } else {
                    for(int x = myServo.getValue() + 10; x <= servo_max;x+=10){
                        myServo.setValue(x);
                        try {
                            Thread.sleep(servo_sleep);
                        } catch (InterruptedException ex) {}
                    }
                }
            } else if(dir == '-'){
                if(myServo.getValue() - amt >= servo_min){
                    int tgt = myServo.getValue()-amt;
                    for(int x = (myServo.getValue() - 10); x >= tgt; x-=10){
                        myServo.setValue(x);
                        try {
                            Thread.sleep(servo_sleep);
                        } catch (InterruptedException ex) {}
                    }
                } else {
                    for(int x = myServo.getValue() - 10; x >= servo_min;x-=10){
                        myServo.setValue(x);
                        try {
                            Thread.sleep(servo_sleep);
                        } catch (InterruptedException ex) {}
                    }
                }
            }
        }
```

The method also keeps the servo from attempting to move beyond its limit points at either the low or the high end. By moving the Servo in steps of 10 each time the movement is smooth and clean, without being either too slow or too fast. By setting the value of servo_sleep (using the associated set method) you can increase or decrease the overall speed of the servo's movement.

Instantiating a Servo

While I have talked about how to control a servo with the above ServoDrive object, I have not specifically shown how to claim the necessary pins on the eDemoBoard to actually drive a servo. One reason for this is that it is so alarmingly easy that making a special section just to cover it seemed superfluous, but it does need to be covered. So, here is the code needed to actually instantiate a Servo object and claim a pin to control a servo:

```
Servo panServo = (Servo)eDemoBoard.getInstance().bindServo(EDemoBoard.H2);
```

Yes, that is all that is required. For the tilt servo, I just change the variable name to `tiltServo` and the pin designation to `EdemoBoard.H1` and I have 2 servo objects claimed and assigned pins and ready to be controlled. I can now pass those servo objects in to the ServoDrive objects and begin.

There are several ways to control the movement of a servo. One is through Pulse Width Modulation (PWM) and the other is through setting the absolute value of the Servo's location. I chose to use the latter method simply because I knew the values of the servo's range, and setting absolute values for the servo's movement was the easiest and most expedient method of controlling the servos for my purposes. You may find that using PWM is a more accurate, more useful or easier method for your purposes and I encourage you to experiment with both methods to determine what works best for your application.

Sun SPOT Radio Connection Software

Creating and maintaining radio communications between Sun SPOTs is almost as easy as instantiating and controlling servos. I chose, in this application example, to use single-hop Radiogram transmissions to communicate between my Sun SPOT BaseStation and the servo-controlling Sun SPOT, but there is no right- or wrong-way to do your radio communications. It is all a matter of what works best for your application.

Again in my application I chose to do bi-directional communications between the BaseStation and the Servo-controlling Sun SPOT so that I could bot set and get the servo's value over the air. Also, I chose once again to have the communications channels be asynchronous by running them each in a separate thread on the BaseStation. I opened a DataGram communications channel to the Sun SPOT BaseStation with:

```
private String SERVO_SERVER_ID = "0014.4F01.0000.024A";
DatagramConnection webServoConn = null;
try {
      webServoConn = (DatagramConnection) Connector.open("radiogram://" +
SERVO_SERVER_ID + ":100");
} catch (IOException ex) { }
```

This simple snippet of code opens a DataGram Connection between the Sun SPOT controlling the Servos (this one) and the BaseStation Sun SPOT which will be sending the Servo commands (identified by it's IEEE address as `SERVO_SERVER_ID` above). Once this communications channel is open, I can then create a RadioGram Object and either read from it, or write data to it and send it to the BaseStation via:

```
Datagram dgIn = null;
try {
   dgIn = myConn.newDatagram(myConn.getMaximumLength());
} catch (IOException ex) {}
myConn.receive(dgIn);
```

To read data from the connection, or:

```
Datagram dgOut = myConn.newDatagram(myConn.getMaximumLength());
int vLocInt = tilt.getLocation();
dgOut.writeInt(vLocInt);
myConn.send(dgOut);
dgOut.reset();
```

To write to it. It is important to remember the `reset()` call to the Datagram object in order to prevent resending the same data repeatedly, and having the Datagram object grow out of control over time until an exception is generated because you have attempted to write beyond the end of the Datagram's size restrictions.

The complete package of code required to run this package can be downloaded from http://sunspot.dev.java.net/ as a jar archive containing all the source code.

Sun SPOT Base Station and Web-integration Software

I wanted to make this project a web-controlled pan-and-tilt camera, so having the controls displayed in an interactive web-page was crucial. While I could have achieved this by having a cgi script communicate from an existing web server to the Sun SPOT BaseStation, I decided to implement at least the bare bones of a web server from the Sun SPOT BaseStation itself. This approach allowed me to do a real bridging application between the Internet and the world of 802.15.4 devices.

To run the webserver end of the application, I simply opened up a socket on port 8888 on the host computer (using the high port number relieved me of the restriction, on my UNIX-based computer, of having to run the process with increased permissions. I did not, however, want to implement everything required to be a complete web server, so my process only handles a specific set of commands.

To make the form, I hard-coded the HTML necessary for the form into `private static final String` objects within the server object itself. The process then simply listens for connections on the open socket, reads the incoming requests, and forwards them to the Servo-controlling Sun SPOT over the radio. Once the message has been sent to the Servo-controlling Sun SPOT, the process then waits for that Sun SPOT to reply with the current location of the Servo so that I can put that value back out in the web-page.

To make sure that the requesting browser understands the response, I just made the first part of the response conformant to what the browser expects with:

```
private static String page1="<!DOCTYPE HTML PUBLIC \"-//W3C//DTD HTML 4.01
Transitional//EN\">" +
            System.getProperty("line.separator") + "<html>" +
              System.getProperty("line.separator") + "<head>" + ...
```

The key was the first portion, telling the receiving browser what was coming. The rest is just standard HTML. I could also have created HTML file fragments and had the process read them from disk and relay their contents back to the receiving browser, but using Strings just seemed more expedient.

Finally, I had to write some methods to parse the incoming messages from the requesting browsers so that I could send appropriate messages to the Sun SPOT controlling the Servos. I chose to use form elements in the web page so that I could easily read and discard all of the browser-based data except the portions contained within the form elements I was interested in using. I could then scan the incoming data for 'left', 'right', 'up', 'down', etc. and for the number to move. Then, just relay that information to the remote Sun SPOT over the radio connection.

Putting it all together

Finally, after all the code was done, and all the wiring was done, I decided to make a reasonably decent-looking mounting system for the entire apparatus. I found a 3” x 3” x 5” rectangular block of white lexan in the machine shop which I used for a base. I drilled out a rectangular hole to accommodate one of the servos and hot-glued it in place. I then used 2 tie-wrap wire ties to attach the second servo, lying flat on its side, to the top of the first servo and I had the pan and tilt axies both covered.

Finally I needed to mount the camera to the servo assembly. To make an arm for the camera i took 2 1” standoffs (used for keeping computer parts apart from one another) and mounted a round-ended wire connector to each end. I was then able to screw one end of the arm to the servo arm, and attach the other end to the camera.

The Apple iSight camera had no easy way to mount the camera to the servos. To accomplish the mounting I found a small strip of scrap aluminum and drilled a hole in each end. I then put a screw through the holes and through a round-end wire connector and tightened the screw so that the aluminum strip was pulled tight around the camera body. I used some more wire ties to secure the camera's cable to the arm assembly, and the camera unit was completed. To secure the Sun SPOT to the device I used the Sun SPOT bracket included in the Sun SPOT Developers Kit by screwing it to the lexan block.

So, what do you have when you're all done? Well, yours will look vastly different, I suspect, but here are a few pictures of what mine looks like:

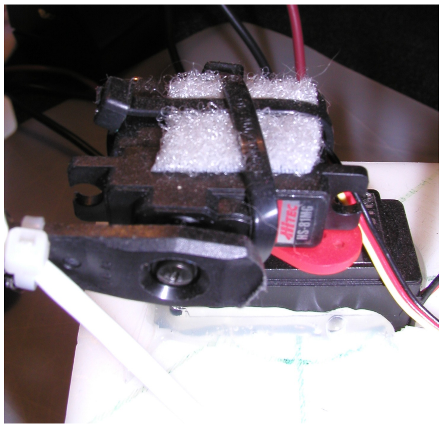


Figure 3: Servos

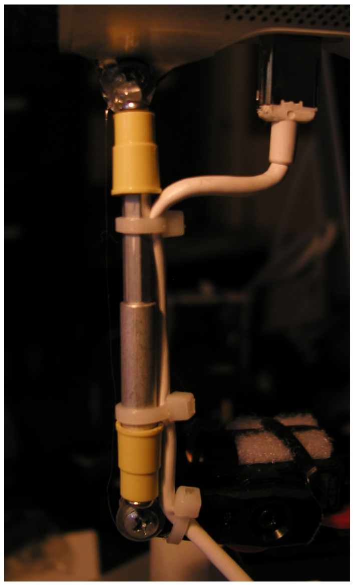


Figure 4: Camera Mount

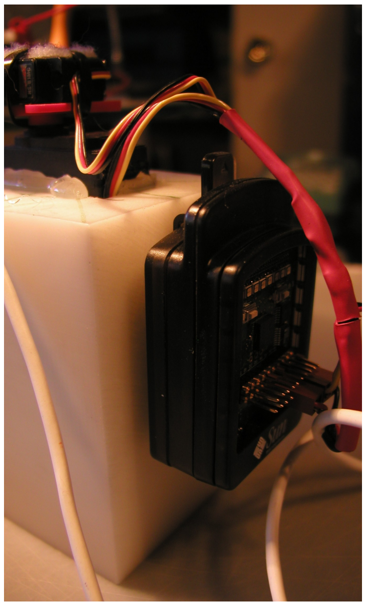


Figure 5: Sun SPOT Mount

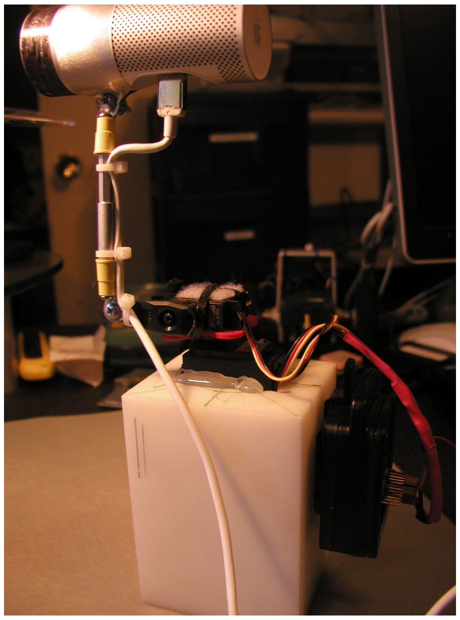


Figure 6: Sun SPOT Pan-and-Tilt Camera Project

About Sun Labs

Established in 1990, Sun Microsystems Laboratories is the applied research and advanced development arm of Sun Microsystems, Inc., with locations in California and Massachusetts. Sun Labs is one of the ways Sun invests in the future, and is responsible for many of the technology advancements that have made Sun a technology powerhouse—including asynchronous and high-speed circuits, optical interconnects, 3rd-generation Web technologies, sensors, network scaling and Java technologies. Although many companies have R&D groups, Sun Labs can claim one of the highest rates of technology transfer in the industry.

Sun Microsystems, Inc. 4150 Network Circle, Santa Clara, CA 95054 USA **Phone** 1-650-960-1300 or 1-800-555-9SUN **Web** sun.com