2-Axis Camera Control with RTLinux/Pro: An Example Project

Cort Dougan

Finite State Machine Labs
Socorro, NM 87801
cort@fsmlabs.com
http://www2.fsmlabs.com/~cort/

Abstract

A low-cost real-time servo motor control system can be easily constructed with RTLinux and a standard PC. In this paper, I'll show how to make a simple and complete system, including a web-based control interface.

1 Overview

RTLinux/Pro allows rapid development of application code and drivers for control of real-world devices. I'm going to show step-by-step how to write all the software necessary to view live images and control a servo-motor driven dual-axis mounted camera via a web-page. While there are important commercial applications for projects like this, anything from security monitors to robotics and manufacturing, I put this camera together so I could watch my cat during the day while I was at the office. My cat, Kepler, is an old guy and has been sick for some time and I want to be able to check on him several times during the day. Kepler usually stays in the same room but I needed to be able to move the camera around to find him in the room. This camera and mount are the perfect solution.

2 Camera Hardware

2.1 Mount

The camera mount is made from scrap pieces of Lexan platic, machine screws and a hose clamp. Fig. 2 shows the mount in a close-up photo. Each axis of the mount is directly driven by a Hobico CS-61 servo motor so that servo motor rotation results in corresponding rotation of the camera. Any type of altitude/azimuth mount with servo motors would work in this example.

Figure 1: Kepler, Worlds Greatest Cat

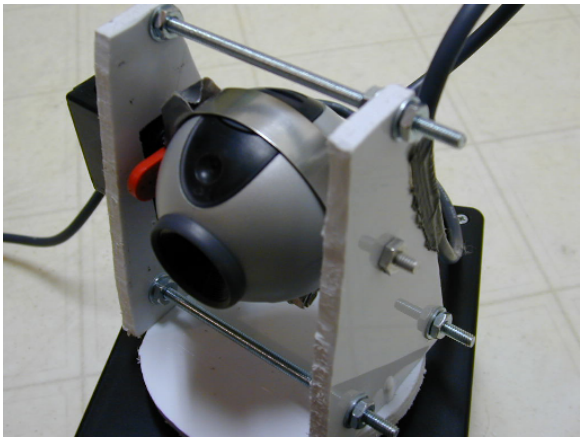

Figure 2: Camera Mount

2.2 Servo Motor Hardware

The servo motors are driven by a simple circuit (Fig. 3) that provides 5v DC, ground and a signal line to each of the motors. There is a DB25 connector on the end that routes the first 5 data lines of the parallel port connector to the 5 motor signal lines. Raising any of these parallel port data lines raises the signal line on that servo motor pin.

2.3 Camera

I used a QuickCam 3000 Pro USB camera for the camera (Fig. 4). It is well supported under Linux, is inexpensive and is able to produce high quality images and produce them quickly (very useful for live video).

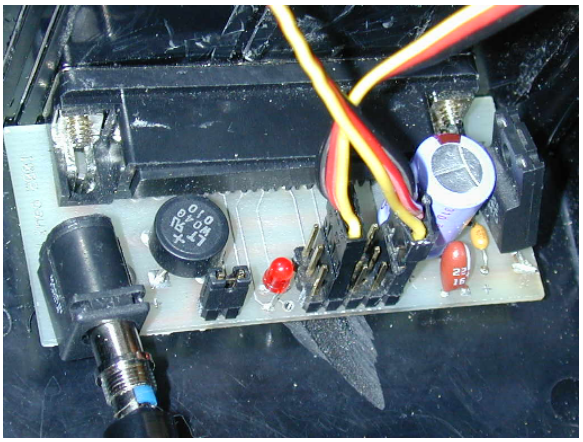

Figure 3: Controller Board

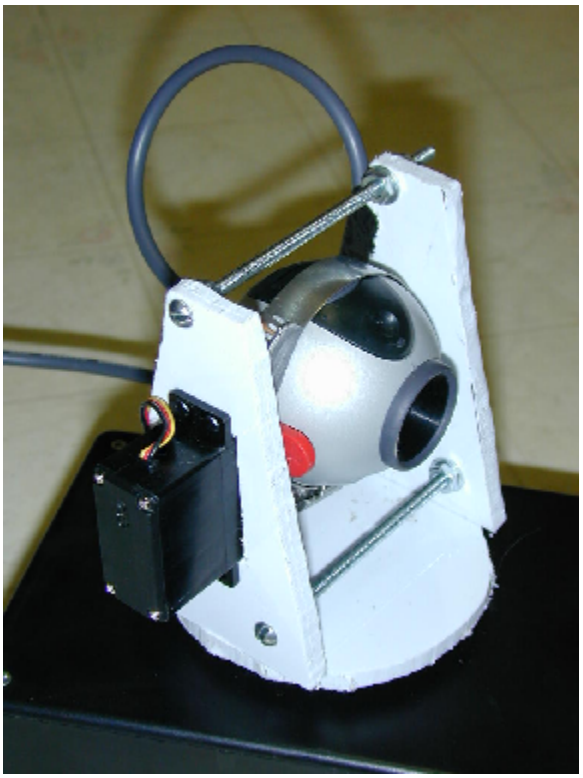

Figure 4: Camera and Mount, with Altitude Servo Motor Visible

3 How RTLinux Applications Work

RTLinux is an operating system that runs Linux as the lowest-priority task. RTLinux applications execute in the RTLinux environment rather than the Linux environment. This gives real-time applications priority over normal Linux applications and even Linux itself. On standard PC's, a 650MHz Pentium III for example, RTLinux can deliver 5μs worst-case interrupt latency and 30μs worst-case jitter for periodic tasks. This means users still have a normal Linux system with X-Windows, web servers, databases and any other application available for Linux but also get an environment where hard real-time applications can run.

RTLinux applications are compiled as loadable kernel modules that are loaded and run after Linux has booted. My goal here is to show how to build, compile and run RTLinux drivers and applications. In order to run these programs you'll need to be running RTLinux. You can find RTLinux on-line at `http://www.fsmlabs.com/products/`. Configuration of RTLinux is described in either in the RTLinux/Free download or the RTLinux Professional distribution and won't be covered here.

First, I'll go through the drivers and application software that load and run in the real-time environment of RTLinux. This is where most of the work of controlling the mount motors is done.

4 RTLinux Parallel Port Driver

In order to drive the motors, through the parallel port connection, I needed to be able to write to the parallel port easily from RTLinux applications. There are parallel port drivers for RTLinux of course, but it's so easy to write one from scratch, that I did.

Drivers in RTLinux advertise their services to other RTLinux applications through files in `/dev`, just as with a standard UNIX system. These `/dev` files that are managed by RTLinux drivers are not directly accessible from the Linux environment. So, a Linux application that opens `/dev/lpt0` is communicating with the Linux (non-realtime) parallel port driver and not the RTLinux driver. Conversely, a RTLinux application that opens `/dev/lpt0` is communicating with the RTLinux driver and not with the Linux driver.

This driver provides a `/dev/lpt0` file that can be used through POSIX `open()`, `read()`, `write()`, `ioctl`, and `close()` calls from RTLinux applications. It makes no distinction between `/dev/lpt0` and `/dev/lpt1`. Instead, it will always write to the same device.

4.1 Driver Initialization

Fig. 5 shows the initialization code for the parallel port driver. The `init_module()` function is called when the module is loaded. The only thing I do in it is call `rtl_register_dev()` to register a handler for `/dev/lpt0`, /dev/lpt1 and so on. The `cleanup_module()` function is called when the driver module is removed and

```
int init_module(void)
{
    if ( rtl_register_dev( "lpt", &rtl_par_fops ) )
    {
        printk("Unable to install driver\n");
        return -EIO;
    }

    return 0;
}

void cleanup_module(void)
{
    rtl_unregister_dev( "lpt" );
}
```

Figure 5: Parallel Port Driver Initialization Code

unregisters the `/dev/lpt` handlers.

4.2 Open and Close

Now, anytime a RTLinux application opens `/dev/lpt0`, the `open()` function that was registered with the call to `rtl_register_dev()` will be called. The same is the case for `close()` calls on `/dev/lpt0` as well. This is the place to do initialization/shutdown of devices or perform any housekeeping necessary on each open/close. Since the parallel port is a simple device and I'm writing a simple driver, none is necessary here. Fig. 6 shows a listing of the open/close code for the driver.

4.3 Read and Write

Most of the work in the driver is done with the `read()` and `write()` calls listed in Fig. 7.

```
static int rtl_par_open(struct rtl_file *filp)
{
    return 0;
}

static int rtl_par_release(struct rtl_file *filp)
{
    return 0;
}
```

Figure 6: Parallel Port Driver Open/Close Code

I assume the value PORT, 0x378, to be the address of the data register of the parallel port. This may vary but is the most common value for PC hardware and makes the driver much simpler. A more full-featured version would have the port address as a configuration value or might even probe for it.

I also maintain the value `out_byte` that stores the last written value to the parallel port. I use this value later when doing `ioctl()` calls.

The `read()` operation is very simple and just needs to read from the data port and returns the value. The first few lines of the function check the input parameters to make sure that enough space was provided to store a single character. The `read()` that this driver implements only reads and returns to the caller a single character even though the user may have provided space for much more data in the argument `buf`. A more full featured driver would likely poll the device until `count` characters were read or would just read until no more data was available on the parallel port. Using the semaphore and signal features in RTLinux it would be easy to make write be more sophisticated, but here we want to treat the parallel port

```
#define PORT 0x378

char out_byte;

static ssize_t rtl_par_read(struct rtl_file *filp,
                char *buf, size_t count,
                off_t* ppos)
{
    if ( count < sizeof(char) )
        return -1;

    buf[0] = inb( PORT );

    return 0;
}

static ssize_t rtl_par_write(struct rtl_file *filp,
                const char *buf,
                size_t count, off_t* ppos)
{
    int i;

    for ( i = 0; i < count; i++ )
    {
        out_byte = buf[i];
        outb( out_byte, PORT );
    }

    return 0;
}
```

Figure 7: Parallel Port Driver Read/Write Code

as a collection of digital IO lines. In fact, my application only wants to output data so I cheated and left `read()` as a placeholder for future updates.

`write()` is not much more complex than `read()`. It loops through each byte of data to be written to the port, saves the output value in `out_byte`, then writes the value to the data register of the parallel port.

4.4 Ioctl

So far I've written code that will allow RTLinux applications for perform `open()`, `read()`, `write()` and `close()`. What happens if we want to have a couple of applications that each need a few of the control lines? For example, in a robot project that I worked on, 2 of the parallel port data lines are used for a camera mount (as in this article) but the other 6 data lines are used for DC drive motors. This driver allows us to do this very easily by creating set-bit and clear-bit operations through `ioctl()` calls.

I added `RTL_PAR_SETBIT` and `RTL_PAR_CLEARBIT` defines to `include/rtl_ioctl.h`. The values of these are unique to each driver so they do not need to avoid conflicting with other drivers.

Fig. 8 lists the `ioctl()` code for our parallel port driver. The `ioctl()` call will either raise (`RTL_PAR_SETBIT` or lower `RTL_PAR_CLEARBIT` the given line (argument `l`) on the parallel port.

```
static int rtl_par_ioctl(struct rtl_file *filp,
            unsigned int request,
            unsigned long l)
{
    switch ( request )
    {
    case RTL_PAR_SETBIT:
        out_byte |= 1<<l;
        break;
    case RTL_PAR_CLEARBIT:
        out_byte &= ~(1<<l);
        break;
    default:
        return -EINVAL;
    }

    outb( out_byte, PORT );

    return 0;
}
```

Figure 8: Parallel Port Driver Ioctl Code

```
include /opt/rtldk-1.1/rtlinuxpro/include/rtl.mk

all: rtl_parallel.o

clean:
    rm -f *.o
```

Figure 9: Parallel Port Driver Makefile

4.5 Compiling the Parallel Port Driver

RTLinux applications must be compiled with very specific options and it's not easy to do by hand. For this reason, RTLinux provides a template Makefile that makes the job nearly trivial.

Fig. 9 shows the Makefile that will build the parallel port driver. The inclusion of `rtl.mk` pulls in all the necessary pre-defined rules and compiler flags necessary to correctly build a RTLinux application. This `Makefile` assumes you've installed RTLinux into `/opt/rtldk-1.1/rtlinuxpro`, the default install location for RTLinux/Pro 1.1. If you install RTLinux (either the Open or Free version) you'll need to update the `Makefile` to represent where RTLinux is installed.

Assuming the source file is named `rtl_parallel.c` the Makefile will build your application with no problems.

5 Servo Motor Driver

Now that the parallel port driver is complete, I can begin with the servo motor driver. The parallel port driver didn't need to communicate with Linux applications (only other RTLinux applications) but the servo motor driver will need to.

The most common communication model between Linux and RTLinux tasks is the realtime FIFO. Anyone who has used a normal FIFO under Linux (as created with `mkfifo`) is familiar with how this works. A process on one end writes to a FIFO, which appears as a normal file, while another one reads from the other end. With RTLinux, the reader might be a realtime process, while the writer is a userspace program shuttling directives to the realtime code through the FIFO, or vice versa. In either case, the FIFO devices are normal character devices (`/dev/rtf*`), and both ends can interact with the devices through normal POSIX calls, such as `open()`, `close()`, `read()` and `write()`.

In this case, Linux tasks will need to be able to send commands to the servo motor driver. I only need user tasks to be able to write to the driver but the servo motor will not need to send messages back to Linux tasks. So I'll use two realtime FIFOs, one for each motor, to send a position that the servo motor should move to.

Once I know where the motor should be, I need to actually move it there. When I initialize the driver, I create one thread (RTLinux task) for each motor. The job of each thread will be to toggle the data line on the parallel port that signals a motor.

To control the Hobico CS-61 motors that I used, I need to send the motor a signal every 20ms. If I want the motor to move to minimum deflection, to the far left, I give it a high on the control line for 1ms and then drop it low for 19ms. For full deflection, to the far right position, I need to raise the signal line for 2ms, then low for 18ms. This gives us about 180deg

of rotation.

Now that I know the general design of the driver, lets look at the code.

5.1 Driver Initialization

In Fig. 10 `init_module()` is called first when our program is loaded. The first thing I do is open the realtime FIFOs that I will use to communicate with Linux tasks. I create a `#define` for, and use, FIFO 16 and 17 corresponding to the first and second servo motors. I picked these specific FIFOs randomly - their numbers have no special significance.

I then install handlers for each of the FIFOs. The first argument to `rtf_create_handler()` selects the FIFO number the handler is installed for. The second argument is a function to be called anytime a Linux tasks does a `write()` to the FIFO. This allows the code to asynchronously read from the Linux side without needing to poll the FIFO.

I then open the parallel port device that I'll be using to control the motors. This `open()` call uses the parallel port driver I described in the previous section.

I then must create the RTLinux tasks that will do the work of turning the parallel port bits on and off. I call `pthread_create()` to create each task. The argument `thread[i]` is where the thread id is stored by the call and `thread_code` is the function to begin executing as the new task. The fourth argument, `i`, is passed as an argument to the function `thread_code` and is used to know which motor to control. The second argument, `NULL`, tells

```
#define SERVO_FIFO 16

pthread_t thread[2];
int fd[2], fd_par;

int init_module(void)
{
    int i;
    char file[256];

    /* open the fifo's */
    for ( i = 0; i < 2; i++ )
    {
        sprintf( file, "/dev/rtf%d",
            SERVO_FIFO+i );
        if ( (fd[i] = open(file, O_RDONLY | O_CREAT |
                O_NONBLOCK) ) < 0 )
        {
            rtl_printf("Could not open %s\n", file);
            return -1;
        }
    }

    /* create FIFO handlers */
    for ( i = 0; i < 2; i++ )
        rtf_create_handler(SERVO_FIFO+i, fifo_handler);

    /* open the parallel port device */
    if ( (fd_par = open("/dev/lpt0",
            O_NONBLOCK)) < 0 )
    {
        rtl_printf("Could not open /dev/lpt0\n");
        return -1;
    }

    /* create the tasks */
    for ( i = 0; i < 2 ; i++ )
    {
        if ( pthread_create( &thread[i], NULL,
                thread_code, (void *)i ) )
            rtl_printf("thread %d failed create\n",
                i);
    }

    return 0;
}
```

Figure 10: Servo Driver Initialization

```
void cleanup_module(void)
{
    int i;

    for ( i = 0 ; i < NUM_MOTORS ; i++ )
    {
        pthread_cancel( thread[i] );
        close(fd[i]);
    }

    close(fd_par);
}
```

Figure 11: Servo Driver Cleanup

`pthread_create()` to use default thread attributes for this task.

`cleanup_module()`, Fig. 11 just does some house-keeping to shut the system down. The `pthread_cancel()` calls stop each task and wait for it to finish. The `close()` calls close the realtime FIFOs and the final `close()` closes the parallel port.

5.2 FIFO Write Handler

Now, lets look at the code that handles a `write()` from the Linux side to one of the realtime FIFOs. In Fig. 10 I registered `fifo_handler()` as the FIFO write handler.

Our handler is called with a single argument that gives us the fifo number that was written to. I use that to `read()` from the proper file descriptor into `msg`. If that was successful, I convert the string in `msg` into an `int` and store it in `position`.

I then test `position` to make sure that it's a sane value, between 0 and 180 degrees of deflection. If there is an error, I return -1. If there is no error, I set `pulse_length` to the time in nanoseconds that I need to raise the output line high.

I command the motor to minimum deflection with a 1ms high signal and full (180deg) deflection with a 2ms high signal. So,

$$pulse\_length = 1ms + \left\{ \frac{1ms}{180\,\mathrm{deg}} * position \right\}$$

I need to avoid doing an actual $\frac{1ms}{180\,\mathrm{deg}}$ since it would lose precision as an integer operation. Since floating point is extremely slow, I should avoid it too. So, I just rely on algebra to save me by doing a $\frac{1ms*position}{180\,\mathrm{deg}}$.

5.3 Pthreads

`thread_code()` is where the motor actually gets moved. This is the code that is timing critical and requires the real-time features of RTLinux. In keeping with RTLinux design principles, this is also the simplest and smallest piece of code so that it's easy to understand and analyze.

The code enters an endless loop terminated only by the `pthread_cancel()` call in `cleanup_module()`. Each iteration of the loop goes through a complete motor command - raising the data line and lowering it.

The first line of code in the loop uses an `ioctl()` call to turn on the bit `num` representing the motor controlled by this task. I must hold the line high for `pulse_length[num]` nanoseconds that was set by `fifo_handler()`.

```
unsigned long pulse_length[2];

int fifo_handler(unsigned int fifo)
{
    int position = -1, err;
    char msg[16];
    char *junk;

    /* read "position" from 0-180 degrees */
    while ( (err = read(fd[fifo-SERVO_FIFO], msg,
                sizeof(msg) )) != 0 )
        position = simple_strtoul(msg, &junk, 10);

    /* stay within the range of the motor */
    if ( (position < 0) || (position > 180) )
        return -1;

    /* compute pulse width */
    pulse_length[fifo-SERVO_FIFO] =
        1000000 /* 1ms */ +
        ((1000000 * position)/180);

    return 0;
}
```

Figure 12: Servo Driver FIFO Code

The `timespec_add_ns()` adds `pulse_length[num]` to the current time and the `clock_nanosleep()` call sleeps until that time has elapsed.

Once I return from the `clock_nanosleep()` call, I'm done holding the line high on the parallel port and now need to lower it. This is timing critical, since each degree of rotation is represented by,

$$\frac{1ms}{180\,\mathrm{deg}} = 5.5\bar{5}\mu s$$

.

difference in the duration of the high signal. Tested under load with a 650MHz Pentium III, RTLinux/Pro gave a worst-case periodic jitter of $30\mu s$ which gives a position accuracy of about $5.4\,\mathrm{deg}$. Linux, without RTLinux, caused delays well over $20ms$ under load when I tested. Linux completely missed frames and would cause the motor to either swing wildly (when holding the line high for incorrect amounts of time) or go completely limp (when missing the frame entirely).

It's possible to optimize periodic timer RTLinux applications (such as this one) down to $0\mu s$ latency with the RTLinux `TIMER_ADVANCE` feature. I've left that out of this example since $5.4\,\mathrm{deg}$ accuracy is enough for this project.

Last in the loop, I call `ioctl()` to lower the data line to the motor, compute how much time is left in the 20ms frame then go to sleep. When I return from the sleep, the loop continues and I do this all over again.

```
unsigned long frame_length = 20000000;

void *thread_code(void *t)
{
    struct timespec next;
    int num = (int)t;

    clock_gettime( CLOCK_REALTIME, &next );

    for (;;)
    {
        /* turn on the pulse */
        ioctl(fd_par, RTL_PAR_SETBIT, num);

        /* setup for the idle part of the duty cycle */
        timespec_add_ns( &next, pulse_length[num] );
        clock_nanosleep( CLOCK_REALTIME,
                TIMER_ABSTIME, &next, NULL);

        /* turn off the pulse */
        ioctl(fd_par, RTL_PAR_CLEARBIT, num);

        /* setup for the next pulse */
        timespec_add_ns( &next,
                frame_length - pulse_length[num] );
        clock_nanosleep( CLOCK_REALTIME,
                TIMER_ABSTIME, &next, NULL);
    }
}
```

Figure 13: Servo Driver Pthreads

```
<HTML>
<HEAD><TITLE>RtlCam</TITLE>
</HEAD>

<FRAMESET COLS="*,60">

<FRAMESET ROWS="*,40">
<FRAME SRC="image.html" NAME="image">
<FRAME SRC="pan.html" NAME="pan">
</FRAMESET>

<FRAME SRC="tilt.html" NAME="tilt">

</FRAMESET>

</BODY>
</NOFRAME></FRAMESET>
</HTML>
```

Figure 14: index.html

6 WebPage

I need the webpage to display the image from the camera and allow me to control rotation and tilt of the camera. The easiest way to do this is through 3 frames, one for each section.

Fig. 14 shows the index webpage that I use to pull all the frames together. This pulls in 3 different HTML files and arranges them properly.

6.1 Image

I use the program `camserv` to display the image from the camera. This program steams images from the camera to UNIX port 9192. This allows any webpage to refer to that port and get a streaming image from the camera without needing to deal with the complexities of manag-

```
<IMG SRC=hostname:9192>
```

Figure 15: image.html

ing the image. I don't go through the details of installing `camserv`, V4Linux or the camera itself since they're all documented very well elsewhere.

Fig. 15 lists the HTML needed to refer to this image once `camserv` is running. Just replace "hostname" with the hostname of the computer that is running `camserv` and the image will appear.

6.2 Camera Control

I'm able to rotate the camera to its full left and then full right stop with:

```
echo 0 > /dev/rtf16
echo 180 > /dev/rtf16
```

Likewise, I can tilt it up and then down with:

```
echo 0 > /dev/rtf17
echo 180 > /dev/rtf17
```

The `pan.html` file, Fig. 16, controls pan position of the camera. It calls a CGI script, `pan.sh`, with an argument that gives it the position to move to. The script just writes this argument to `/dev/rtf16` which actually moves the camera. You'll notice that I refer to positions $-90\,\mathrm{deg}$ through $90\,\mathrm{deg}$ instead of 0 through 180 on the webpage. This seems to make more sense for the end user even though I represent it differently internally.

```
Camera Position, relative to center:
<a href="/cgi-bin/pan.sh?0">-90</a>
<a href="/cgi-bin/pan.sh?15">-75</a>
<a href="/cgi-bin/pan.sh?30">-60</a>
<a href="/cgi-bin/pan.sh?60">-30</a>
<a href="/cgi-bin/pan.sh?75">-15</a>
<a href="/cgi-bin/pan.sh?90">center</a>
<a href="/cgi-bin/pan.sh?105">+15</a>
<a href="/cgi-bin/pan.sh?120">+30</a>
<a href="/cgi-bin/pan.sh?135">+45</a>
<a href="/cgi-bin/pan.sh?150">+60</a>
<a href="/cgi-bin/pan.sh?165">+75</a>
<a href="/cgi-bin/pan.sh?180">+90</a>
```

Figure 16: pan.html

I do something very similar with tilting the camera, Fig. 17.

7 Further Application

This simple project shows how most RTLinux projects can be designed and completed. There is always a small, timing critical, realtime piece of code and a number of user-level applications tied together. The goal is always to keep the realtime portion small and simple to allow it to execute in the least amount of time and with the greatest determinism.

Writing drivers and applications is simple and easy. It can be done through standard POSIX calls and concepts. Even communication between Linux and RTLinux tasks, to for cooperative applications, can be done with POSIX calls.

```
Tilt:
<a href="/cgi-bin/tilt.sh?180">+90</a><br>
<a href="/cgi-bin/tilt.sh?165">+75</a><br>
<a href="/cgi-bin/tilt.sh?150">+60</a><br>
<a href="/cgi-bin/tilt.sh?135">+45</a><br>
<a href="/cgi-bin/tilt.sh?120">+30</a><br>
<a href="/cgi-bin/tilt.sh?105">+15</a><br>
<a href="/cgi-bin/tilt.sh?90">center</a><br>
<a href="/cgi-bin/tilt.sh?75">-15</a><br>
<a href="/cgi-bin/tilt.sh?60">-30</a><br>
<a href="/cgi-bin/tilt.sh?45">-45</a><br>
<a href="/cgi-bin/tilt.sh?30">-60</a><br>
<a href="/cgi-bin/tilt.sh?15">-75</a><br>
<a href="/cgi-bin/tilt.sh?0">-90</a><br>
```

Figure 17: tilt.html

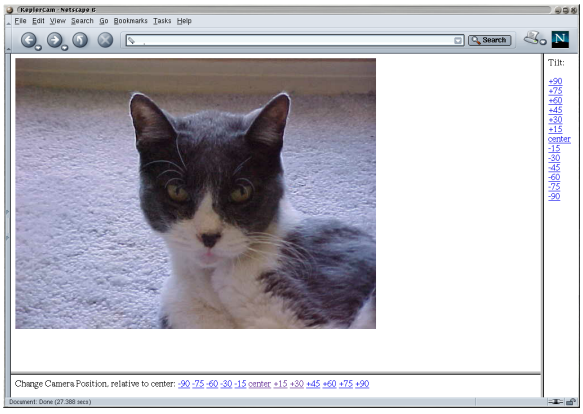

Figure 18: Screen Shot of the Webpage