SPARK Museum Mechanical Television Exhibit

Mark Allyn, Bellingham, Washington

Overview

The SPARK Museum of Electrical Invention owns two Nipkow disk mechanical television receivers. One is a kit receiver that is sold in parts and is used for experimentation and is not meant to be used for entertainment. The other is a Baird Televisor, which was a British product meant to be used for entertainment.

During the era of mechanical television, these receivers used signals that were sent by AM broadcast radio. Those signals were amplified and used to modulate the brightness of a large aperture neon bulb. The bulb is viewed through the holes in a Nipkow disk. The Nipkow disk has a spiral pattern of holes drilled around the circumference of the edge.

Those signals were derived from a mechanical television camera. For the Baird televisor system, that camera consisted of a darkened room in which the subject is scanned with a beam of light and the returned light is picked up by photocells, amplified, and then sent to a radio broadcast transmitter.

Since the museum does not have a mechanical television camera, we need a means of providing signals in order to demonstrate the two mechanical television receivers.

I decided to use a combination of computer software, digital hardware, and finally anlog hardware to simulate the mechanical television signals.

There is another objective as well. During the era of mechanical television, the synchronization between the scanning of the subject by the mechanical television camera and the rotation of the Nipkow disk in the receiver was very difficult and often required constant adjustment of the disk's rotation speed and phase by the viewer. We did not want to imposed that to the museum visitor, so I needed to create a feedback mechanism to synchronize the signal to the actual speed and rotational phase of the spinning disk.

However, I did decide to provide an option for museum visitors to experience trying to achieve manual synchronization for the Baird mechanical television, which does have a manual disk speed control. One of the pushbuttons on the video signal processor turns automatic synchronization on and off.

The solution I came up with uses:

1. Raspberry PI Single Board Computer; a popular development processor that has a large support community of electronic hobbyists. There is one of these to create the digital video signal for both the kit and the Baird receivers.
2. Infrared LED and photodiode detector to detect timing of the holes in the spinning disks. There is one of these mounted inside of each the kit and the Baird receivers.

3. Resistor ladder based digital to analog conversion to convert the digital video signals from the Raspberry PI to analog signals. There is one of these for each the kit and Baird receivers.
4. Low level Op Amp based analog signal conversion to convert the output from the resistor ladder analog to digital converter to appropriate signals for the high voltage high current N Channel MOSFET. There is one of these for each the kit and the Baird receivers.
5. High voltage and high current N-Channel MOSFET transistor. This transistor controls current based on a voltage to the gate. There is one of these for each the kit and the Baird receivers.
6. Combination infrared LED and Photodiode (and associated electronic circuit) that will create a hardware synchronization pulse that is aligned with the rotation of the mechanical television receiver's spinning disk. There is one of these mounted inside each the kit and the Baird receiver.

Note on light source

During the era of mechanical television, the light source used in many mechanical television receivers was a large aperture neon bulb.

These neon bulbs were not very bright, which required mechanical television to be viewed in a very dimly lit room.

This is especially true for the Baird mechanical television because the holes in the Nipkow disk are very small. After I have done some testing, I had discovered that the Baird mechanical television would have to be viewed in an almost completely dark room.

Since ambient lighting is required in a public museum setting, I chose to substitute a much brighter LED array for the neon bulb in the Baird mechanical television.

This would allow museum guests to comfortably witness an operating Baird mechanical television.

The holes in the Nipkow disk for the kit mechanical television, however are much larger than those on the Baird mechanical television.

There is barely enough light transmitted through the holes of the spinning Nipkow disk in the kit mechanical television to allow viewing with ambient indoor lighting. I decided to use the existing neon bulb in the kit mechanical television.

By using the neon bulb in the kit mechanical television, I am hoping to demonstrate to museum guests the experience of watching period mecanical television as it was with a neon bulb.

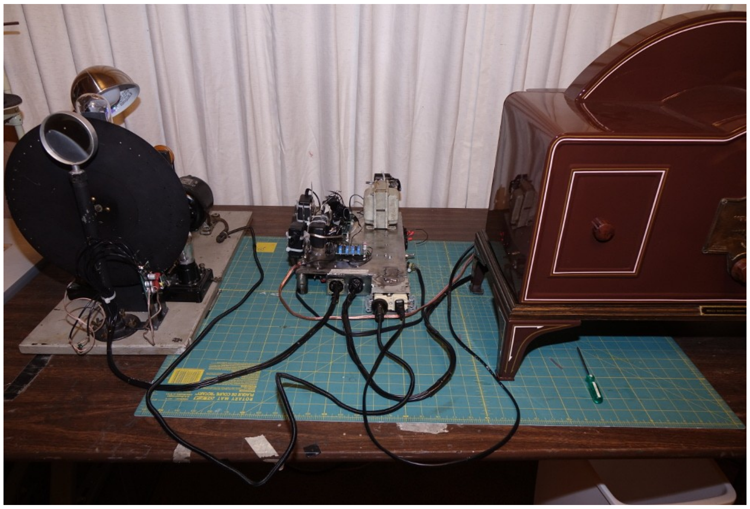
Interconnection

There are two connections between the kit mechanical television and the video signal processor. They are the signal cable and the electric power cable to the motor.

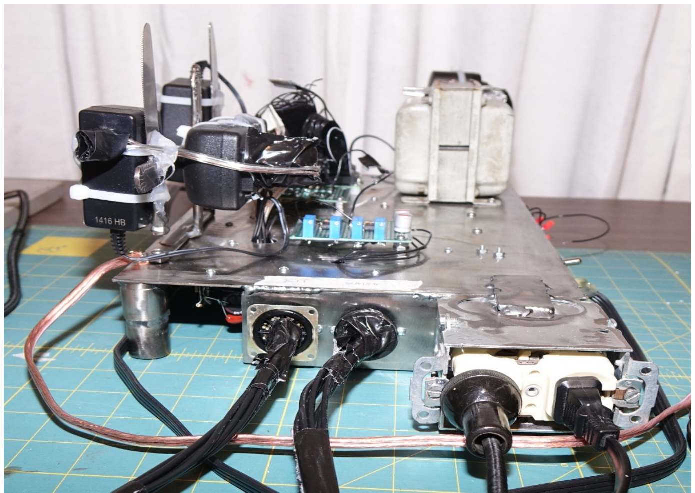
There are three connections between the Baird mechanical television and the video signal processor. They are the signal cable; the electric power cable to the motor; and a grounding cable.

There is a terminal strip with connections for three LEDs, and two pusbuttons that can be installed on an external cabinet for the signal processor.

This is the overall view of the video signal processor along with the two mechanical television receivers. The kit mechanical television is on the left and the Baird Televisor is on the right.



Following is a close up of the connectors on the edge of the signal processor chassis:

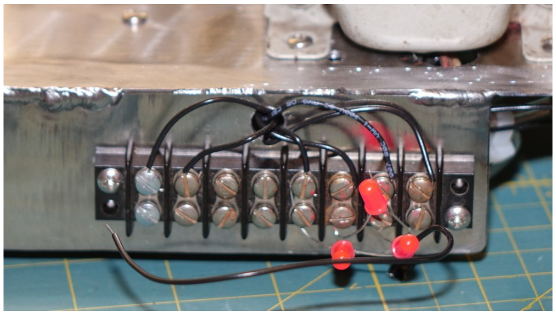


The two multi pin octal electrical sockets in the center are for the timing and video signals; the one on the left is for the kit mechanical television and the one on the right is for the Baird Televisor. T

he two standard electrical outlets are for the mechanical television receivers' motors.

The ground for the Baird mechanical television is the cable that is connected directly to the surface of the chassis to your left.

The following is a close up of the terminal strip that is on the right hand side apron of the chassis:



Looking from left to right, the terminals are:

1. Pushbutton to activate the mechanical television receivers
2. Pushbutton to switch between automatic synchronization and manual synchronization for the Baird mechanical television
3. Not Used
4. Not Used
5. LED indicating Baird manual synchronization
6. LED indicating mechanical Televisions are running
7. LED indicating device is powered on
8. Ground return for all LEDs and pushbuttons

Discussion of Processor Operation

The goal of the signal processor is to convert the video data in an animated GIF file (a type of computer video file) to the necessary electrical pulses needed by both the neon bulb in the kit mechanical television and the LED array in the Baird mechanical television. In addition to converting the video signal from digital (as they are in the GIF files) to analog, the following need to be done:

1. Convert the resolution of the image to the required resolution for each the kit and the Baird mechanical televisions
    1. Animated GIF file is 512 pixels horizontal and 512 pixels vertical
    2. Baird mechanical television is 30 pixels horizontal and 40 pixels vertical
    3. Kit mechanical television is 28 pixels horizontal and 48 pixels vertical
2. Convert color video signal to monochrome
3. Convert the Cartesian coordinates for each pixel from the GIF file to those for each the kit and Baird mechanical televisions
4. Using feedback from each mechanical television; synchronize the output for each pixel and line of the image
5. Convert the low level (3 volt) analog video signals to appropriate current and voltage for each of the mechanical televisions:
    1. 15 volts at between 0 and 100 milliamperes for the LED array in the Baird mechanical television
    2. 250 volts at between 0 and 20 milliamperes for the Neon bulb in the kit mechanical television

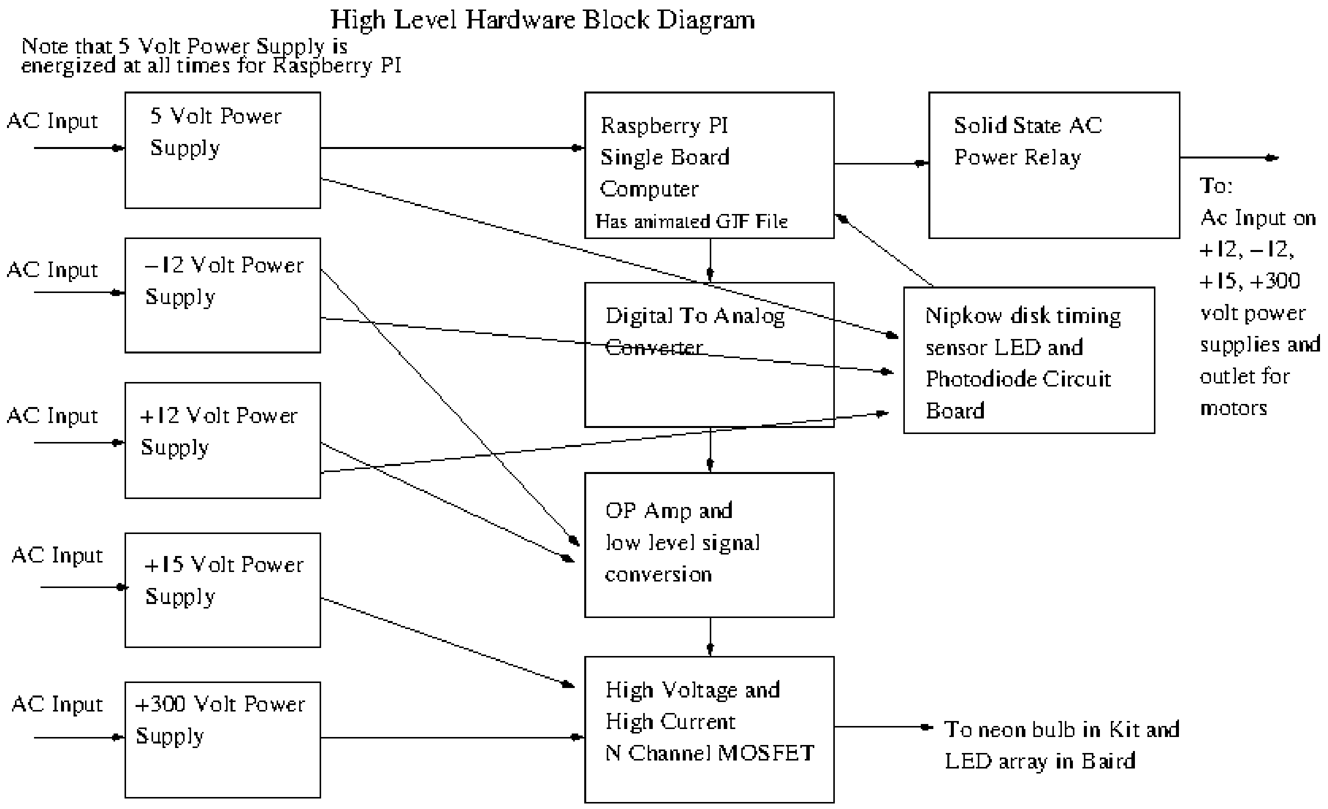
The conversion of the image resolution; the Cartesian coordinate conversion; and synchronization functions are all performed by software running of the Raspberry PI single board computer.

The Raspberry PI single board computer furnishes a 4 bit (16 level) digital video signal for each the kit and the Baird mechanical televisions.

The timing signals are furnished to the Raspberry PI from infrared photodiode circuits mounted in each the kit and Baird mechanical televisions.

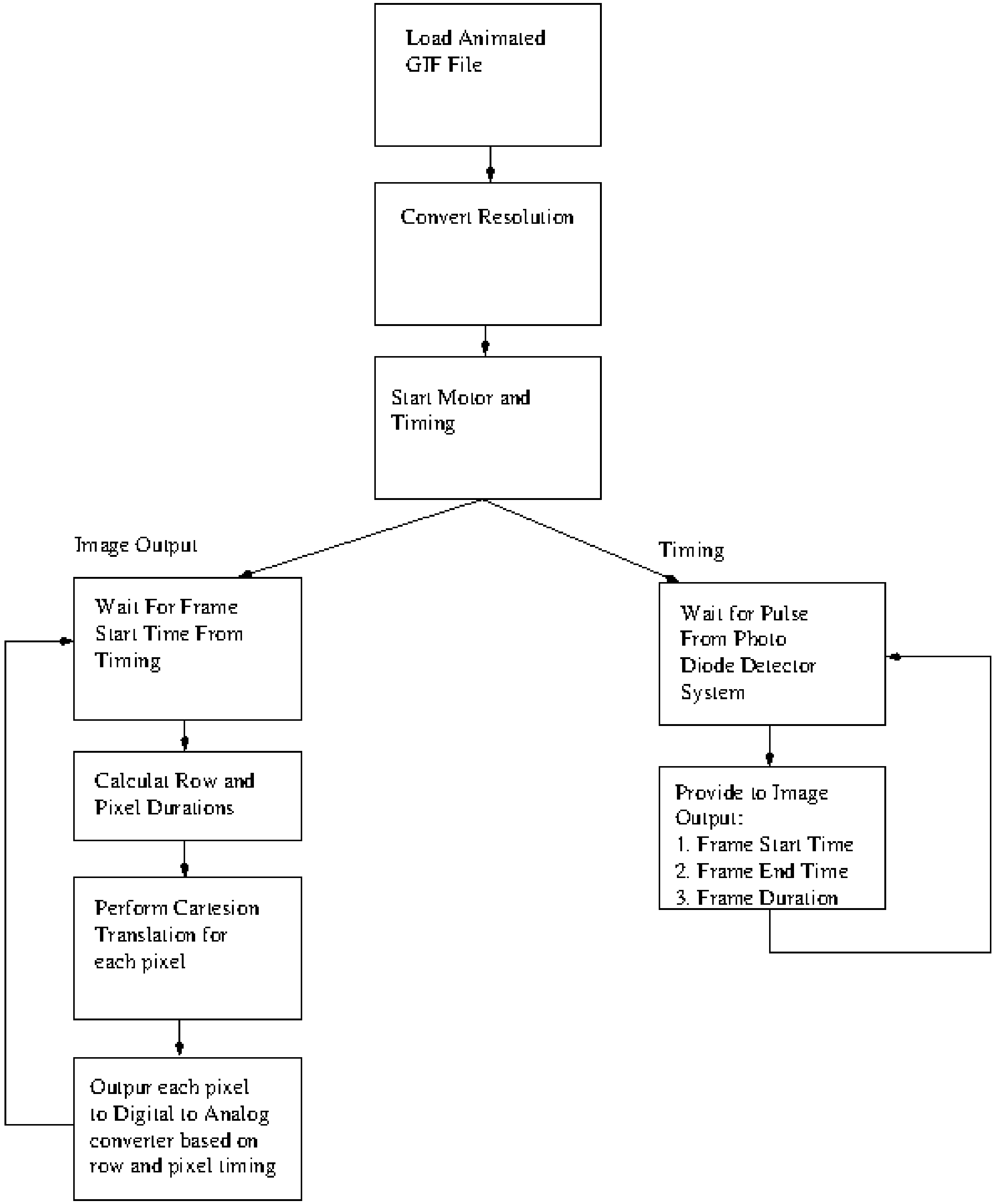
The analog signal conversion is performed on circuitry on the signal processor chassis before the signals are provided to the mecanical televisons.

The overall system block diagram is follows:


High Level Hardware Block Diagram
Note that 5 Volt Power Supply is energized at all times for Raspberry PI
AC Input
5 Volt Power Supply
AC Input
−12 Volt Power Supply
AC Input
+12 Volt Power Supply
AC Input
+15 Volt Power Supply
AC Input
+300 Volt Power Supply
Raspberry PI Single Board Computer
Has animated GIF File
Digital To Analog Converter
OP Amp and low level signal conversion
High Voltage and High Current N Channel MOSFET
Solid State AC Power Relay
To: Ac Input on +12, −12, +15, +300 volt power supplies and outlet for motors
Nipkow disk timing sensor LED and Photodiode Circuit Board
To neon bulb in Kit and LED array in Baird

Here is a very high level flow for the software running on the Raspberry PI that gets the timing puleses from the photodiode circuit in each of the mechanical television and then outputs the digital image in proper synchronization:

High Level Flow


Load Animated GIF File
Convert Resolution
Start Motor and Timing
Image Output
Timing
Wait For Frame Start Time From Timing
Calculat Row and Pixel Durations
Perform Cartesion Translation for each pixel
Outpur each pixel to Digital to Analog converter based on row and pixel timing
Wait for Pulse From Photo Diode Detector System
Provide to Image Output:
1. Frame Start Time
2. Frame End Time
3. Frame Duration

Timing

The disk revolution detector (the infrared LED and Photodiode assembly in each the Baird and the Kit) issues one pulse for each revolution of the disk. The pulse is issed at the same relative position of the holes to the arpiture and lamp.

The pulse is sent to a GPIO pin for each the Kit and the Baird. That pulse will cause an interrupt to be issued to the Raspberry PI processor. The interrupt causes a function to be executed that does:

1. Save the starting time for the image (which can be adjusted by the delay value stored in the software
2. The duration of the image (in nanoseconds)

The video processing power constantly monitors the clock. When the starting time for an image has arrived, the processor does the following:

1. Calculate line and pixel durations
2. Loop for each row:
    1. See if we are in new frame (compare frame start time from clock; if it has advanced, we need to stop processing this frame and start on new frame
    2. Wait for row start time
    3. Loop for each pixel:
        1. Wait for pixel start time
        2. Perform cartesion coordinate calculation for that pixel based on line and pixel count
        3. Output pixel to digital to analog converter
        4. Update pixel start time for next pixel using pixel duration
    4. Update row start time based on row duration
3. Wait for next image startin

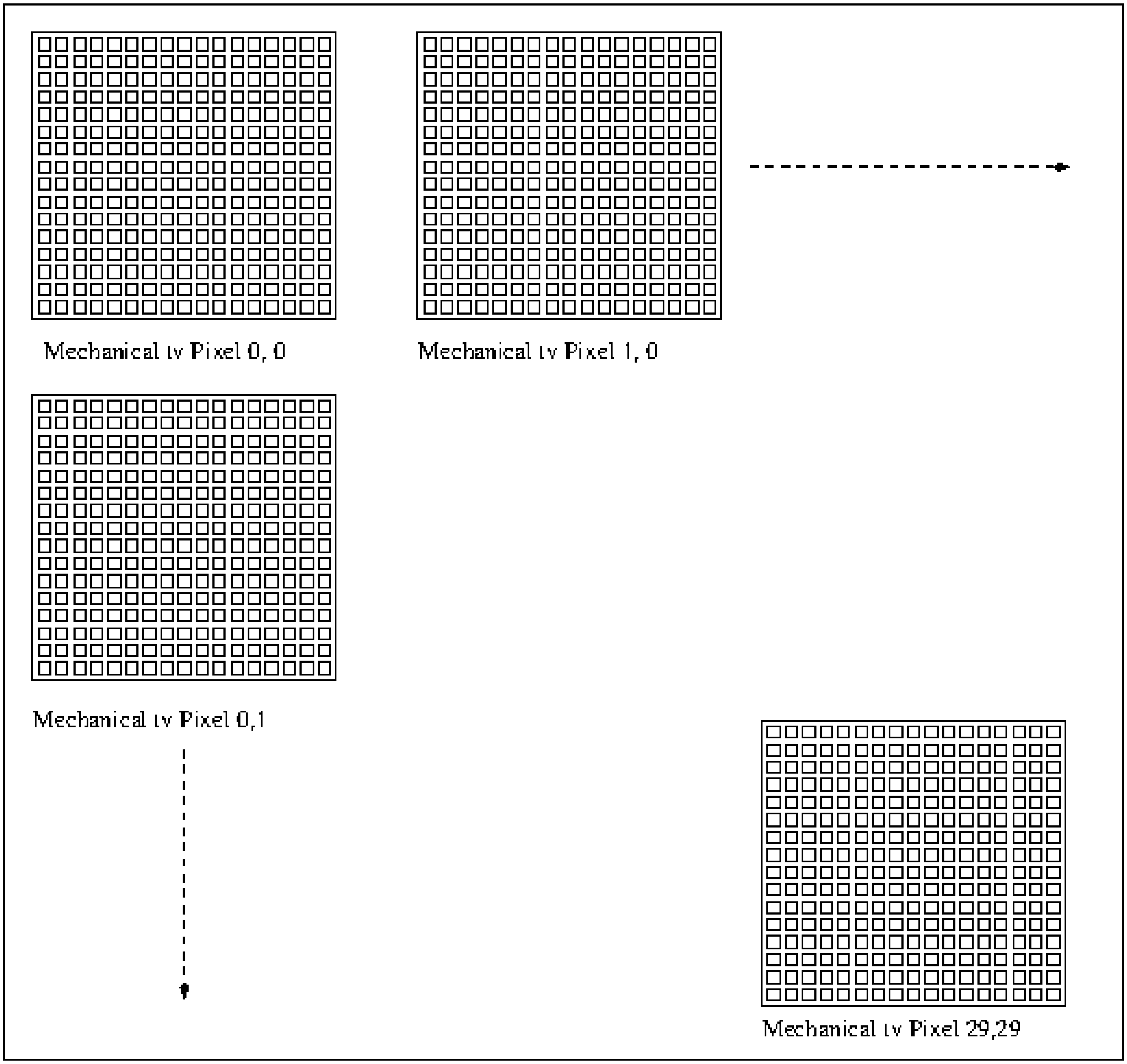
Discussion of Digital Image Conversion

The first step performed by the software is to load the entire animated GIF file into memory.

After loading the image into memory, the image is converted to the proper resolution for each the kit and the Baird mechanical televisions. This means reducing the number of pixels in both the vertical and horizontal resolution.

Each of the pixels for mechanical television represents a block of pixels from the GIF file (since there are far fewer pixels in the mechanical television image than that of the GIF file). The following diagram illustrates this: (Please note that I am using an example mechanical television resolution of 30 by 30 pixels; the kit and Baird are different, but the principals are the same)

Each GIF image is 512 by 512 pixels. Each sample mechanical tv image is 30 by 30 pixels. Therefore, each sample mechanical tv pixel represents a block of 17 GIF pixels

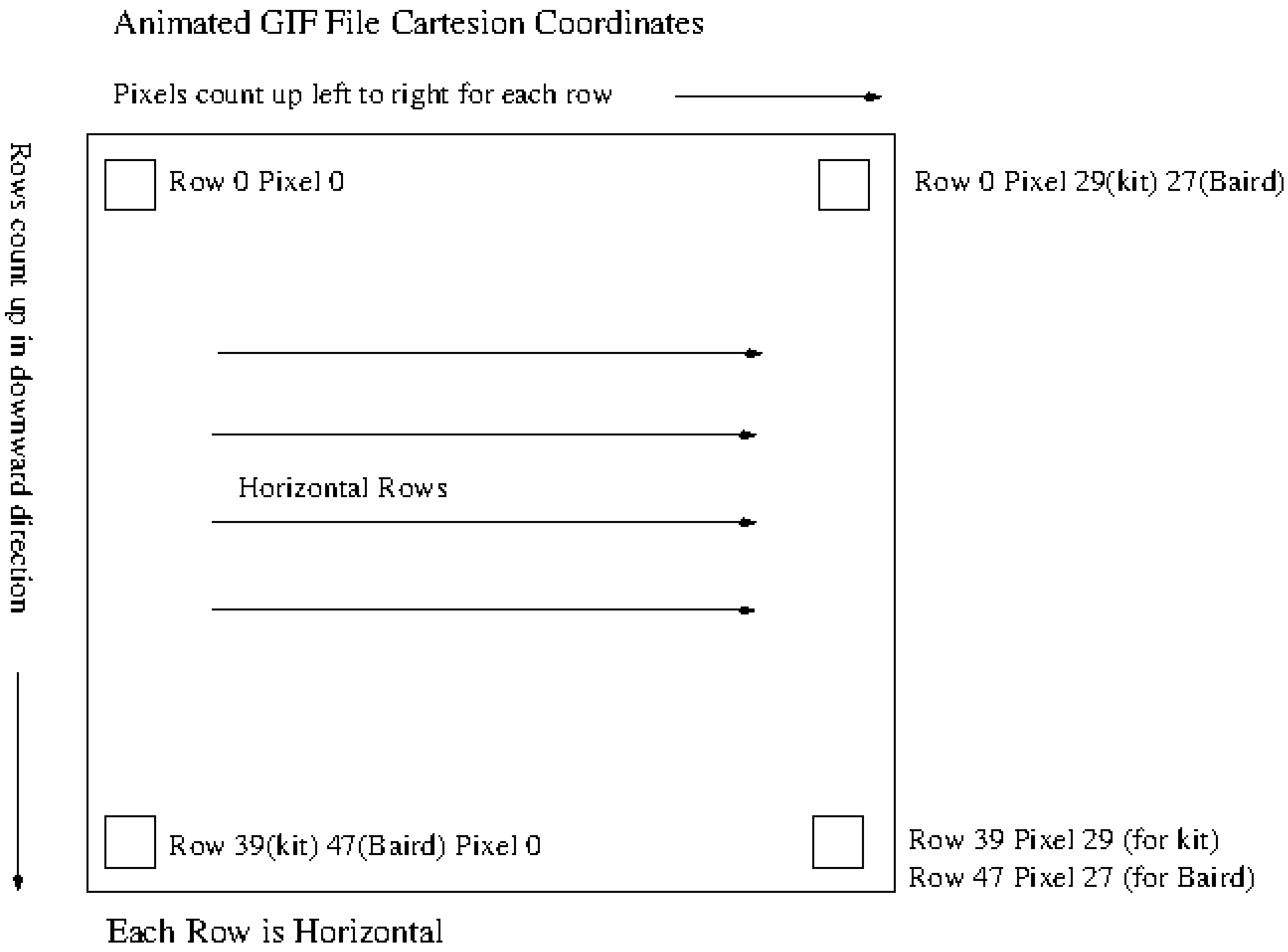


After each image is converted for size, the next step is cartesion axis conversion. In a GIF file, the images are arranged so that the pixels start at the upper left hand corner of the image and proceed horizontally from left to right for each line. The lines in turn count up from top of image to the bottom.

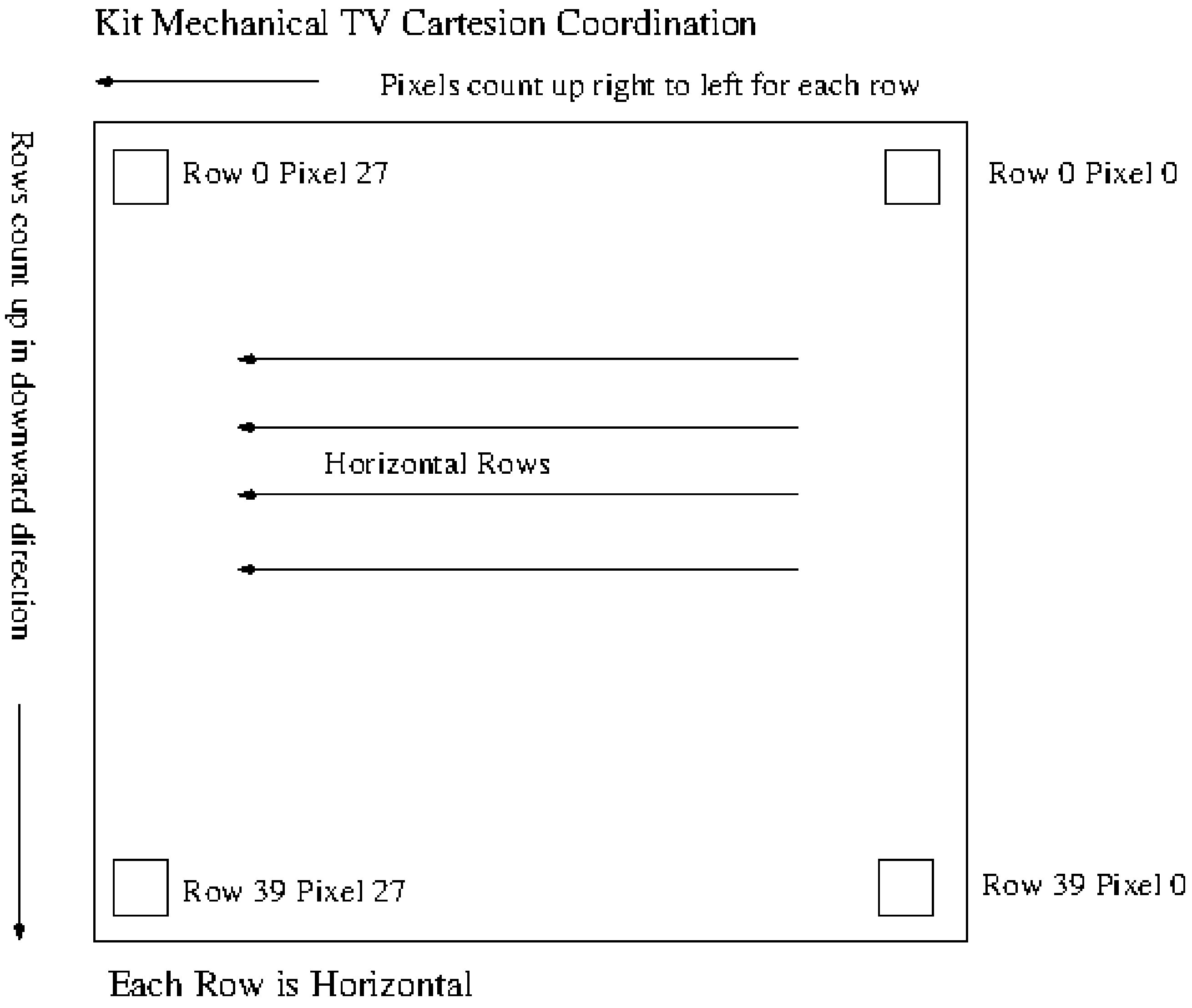
For the kit mechanical television, the pixels start at the upper right hand corder of the image. That means that the count is from right to left for each line of the image. The lines still count up from top of image to the bottom of the image.

The Baird has the most changes to the cartesion coordinates. The image starts at the lower right hand corner. Each line is vertical; the pixels count up from the bottom of the image to the top of the image. The lines in turn, count up from the right hand side of the image to the left hand side of the image.

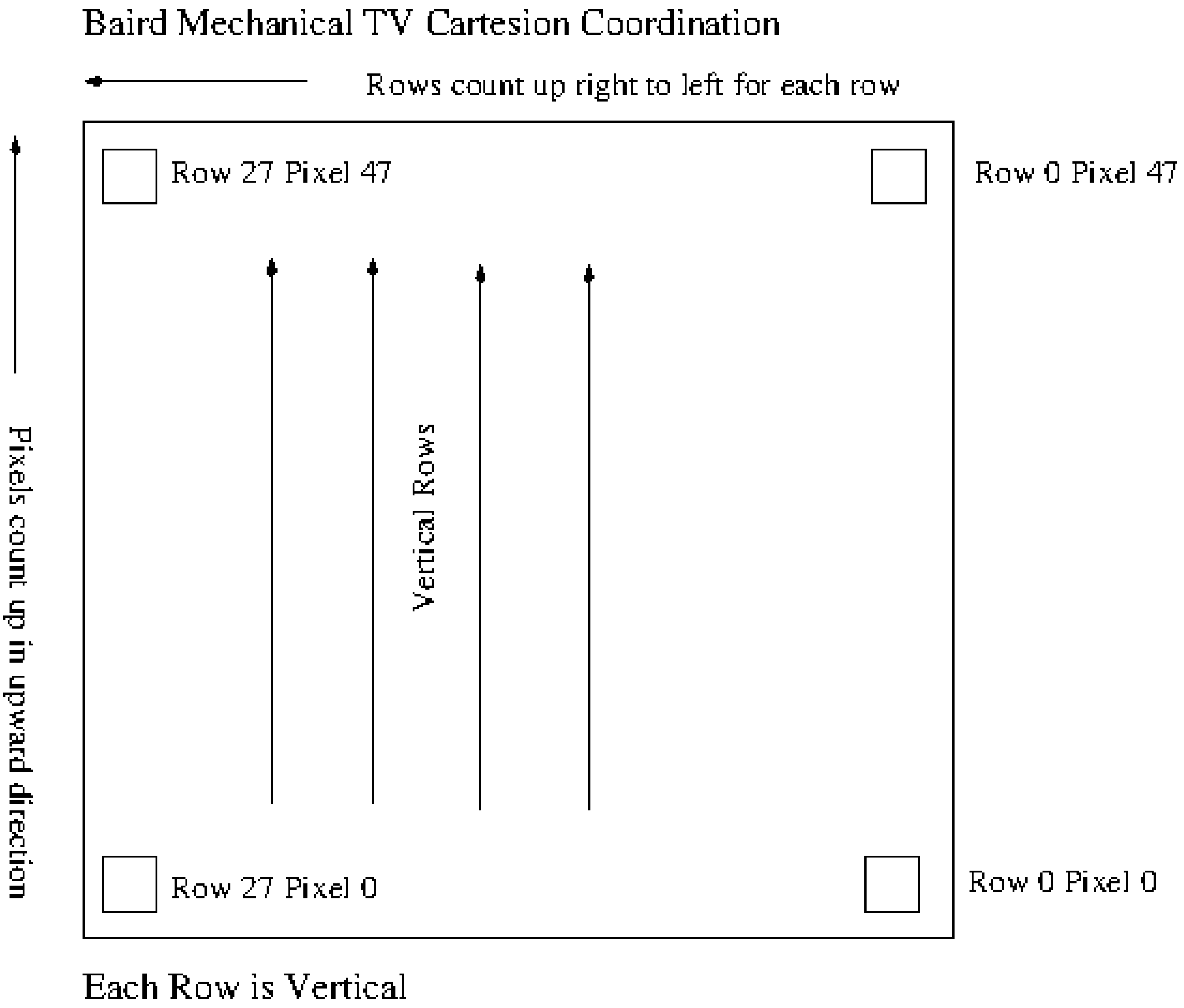
Hopefully, the following images will help illustrate the cartesion coordinate conversion:


Animated GIF File Cartesion Coordinates
Pixels count up left to right for each row
Rows count up in downward direction
Row 0 Pixel 0
Row 0 Pixel 29(kit) 27(Baird)
Horizontal Rows
Row 39(kit) 47(Baird) Pixel 0
Row 39 Pixel 29 (for kit)
Row 47 Pixel 27 (for Baird)
Each Row is Horizontal

The first image shows the coordinates for the GIF file


Kit Mechanical TV Cartesion Coordination
Pixels count up right to left for each row
Row 0 Pixel 27
Row 0 Pixel 0
Rows count up in downward direction
Horizontal Rows
Row 39 Pixel 27
Row 39 Pixel 0
Each Row is Horizontal

The second image shows the coordinates for the Kit mechanical television. Therefore, as you scan from right to left on the kit, row 0 pixel 0 on the kit is row 0 pixel 27 of the GIF image; row 0 pixel 1 on the kit is ros 1 pixel 26 on th e GIF image and so forth. The row count is the same.


Baird Mechanical TV Cartesion Coordination
Rows count up right to left for each row
Row 27 Pixel 47
Row 0 Pixel 47
Vertical Rows
Pixels count up in upward direction
Row 27 Pixel 0
Row 0 Pixel 0
Each Row is Vertical

The third image shows the coordinates for the Baird mechanical television. Here we have to do a double backward count. Baird scans from bottom to top and right to left. Row 0 pixel 0 (start of scan) is Row 27 Pixel 47 on the GIF image. Row 0 pixel 1 is Row 27, pixel 46). Likewise, Row 1 pixel 0 (next row) would be row 26 pixel 47 of the GIF image.

These coordinate conversions are performed by the software as it prepares to output each pixel's digital video data to the digital to analog converter.

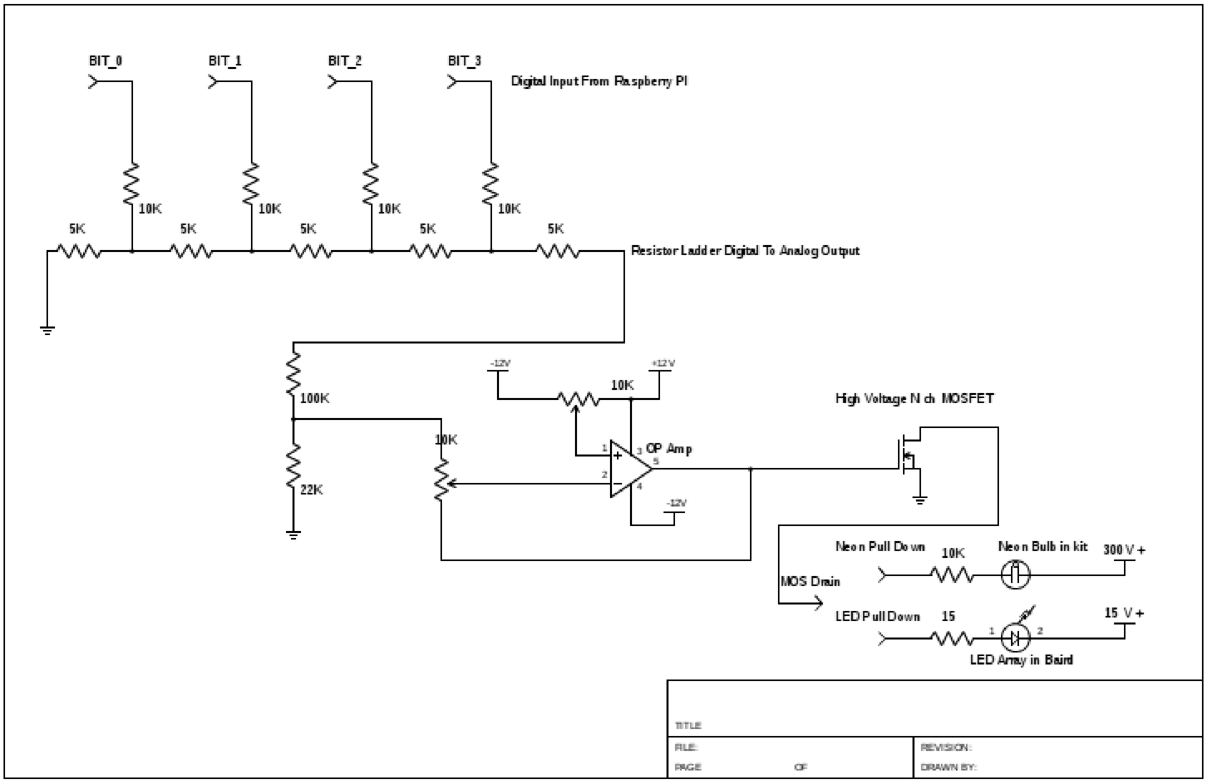
Discussion of analog signal generation and conversion

For each the kit and the Baird, the Raspberry pi and the software will output a 4 bit value for each pixel for each mechanical television.

The four bit binary data uses a 0 volt logical 0 and 3 volt logical one for binary values.

This 4 bit binary value is issued to a resistor ladder digital to analog converter.

The output of the digital to analog converter is a 1 – 3 volt analog value for the video signal. That signal is amplified by an op amp. The op amp's output is then sent to a high voltage and high current MOSFET. The MOSFET in turn controls the current going throught the neon bulb (for the kit) and the high power LED array (for the Baird).



The resistor ladder is at the upper left hand corner of the schematic. There was no real logic in selecting the resistor values for the ladder. I merely found a bunch of resistors at the Spark Museum's workshop and used them. The only criteria was that the resistors from each of the bit signals are supposed to be twice the value for those in the series string from ground to the output.

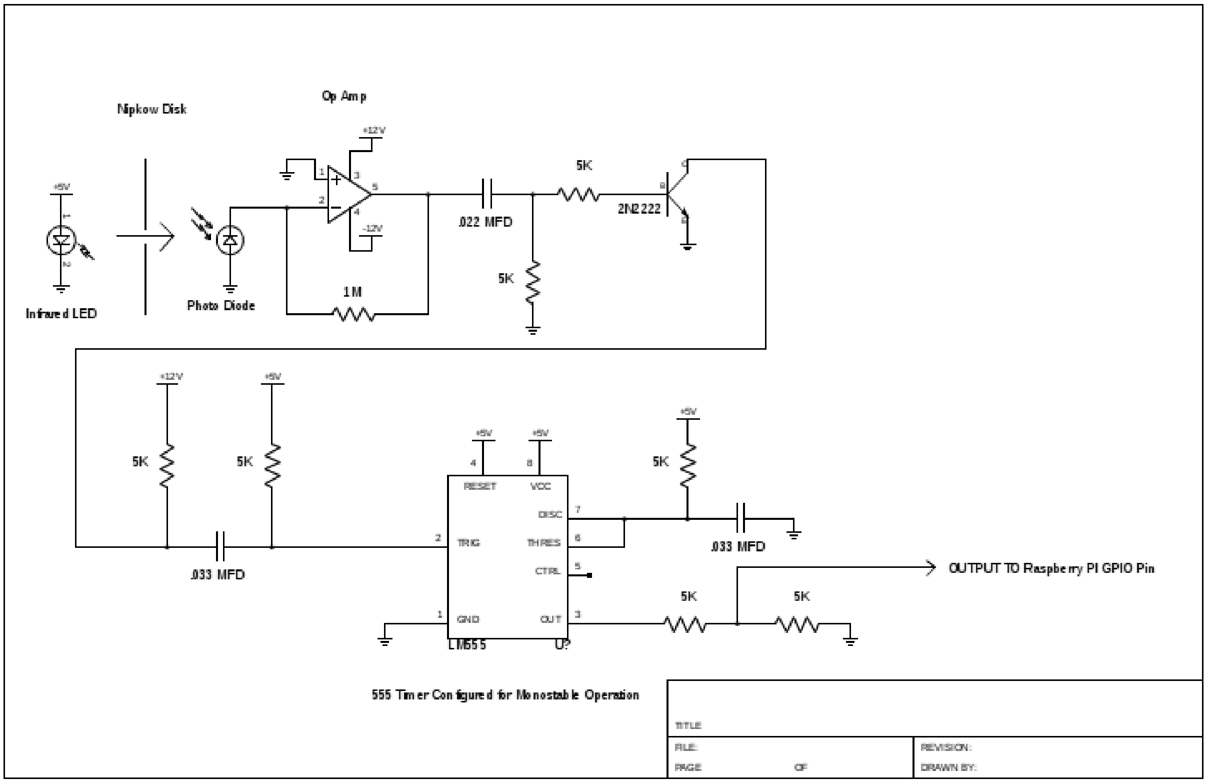
Please note there are two variable resistors with the Op Amp. One of them is to adjust the gain of the

Op-Amp and the other is to adjust the voltage offset. I chose to use variable resistors because I feel that either the neon bulb or the led array may drift (especially the neon bulb) and the the offset/gain may need to be re-adjusted if the gamma of the video (gray scale) changes over time.

Please not that this schematic is duplicated, one each for the Kit mechanical television and the Baird mechanical television. All of the components with the exception of the current limiting resistor connected to the drain of each of the MOSFETs are identical.

Timing Sensor Schematic

The timing sensor (the infrared LED and PhotoDiode) circuit is identical for both the Kit and Baird mechanical televisions.

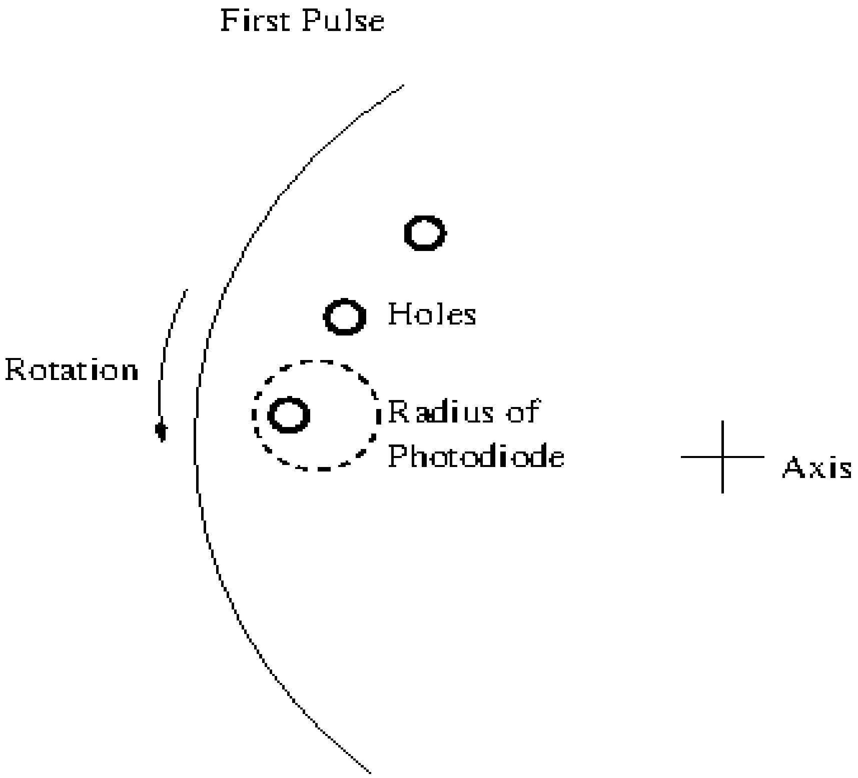


The LED and PhotoDiode are at the upper left hand corner of the schematic. The photodiode feeds an OP-Amp which provides current gain. The OP-Amp feeds a 2N2222 transistor that inverts the signal. The inversion is required because the LM555 timer requires a negative going pulse and the OP-Amp puts out a positive going pulse.

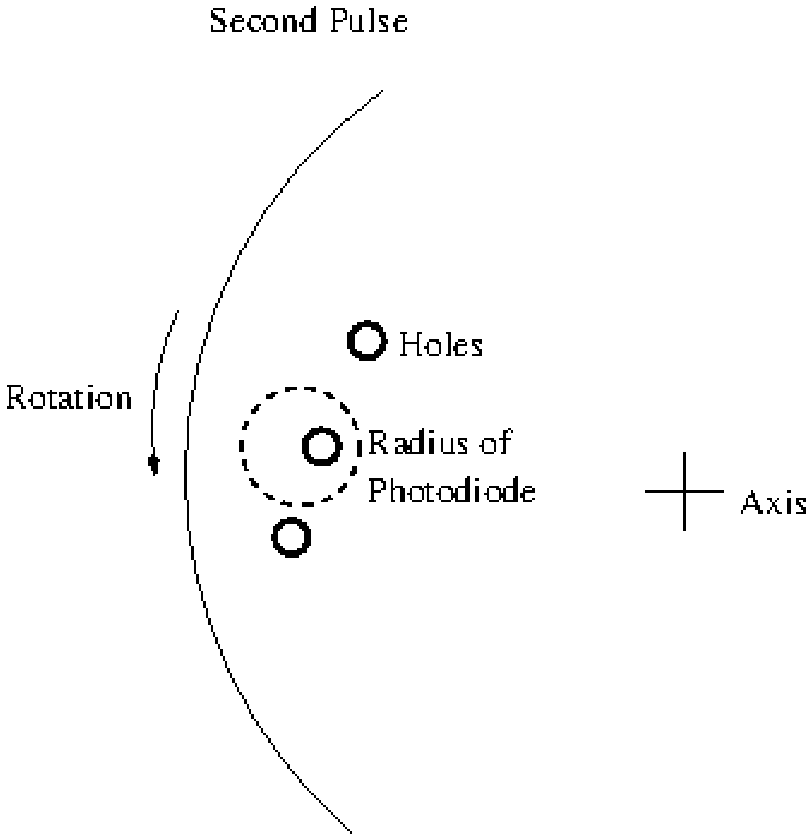
The LM555 timer is configured for monostable multivibrator operation.

The following discussion addresses why I had to add a multivibrator to the timing sendor.

Please note the following figure. This shows (in greatly simplified form) the relationship of the holes in the Nipkow disk and the photodiode and LED when the first hole passes the aparture of the PhotoDiode:

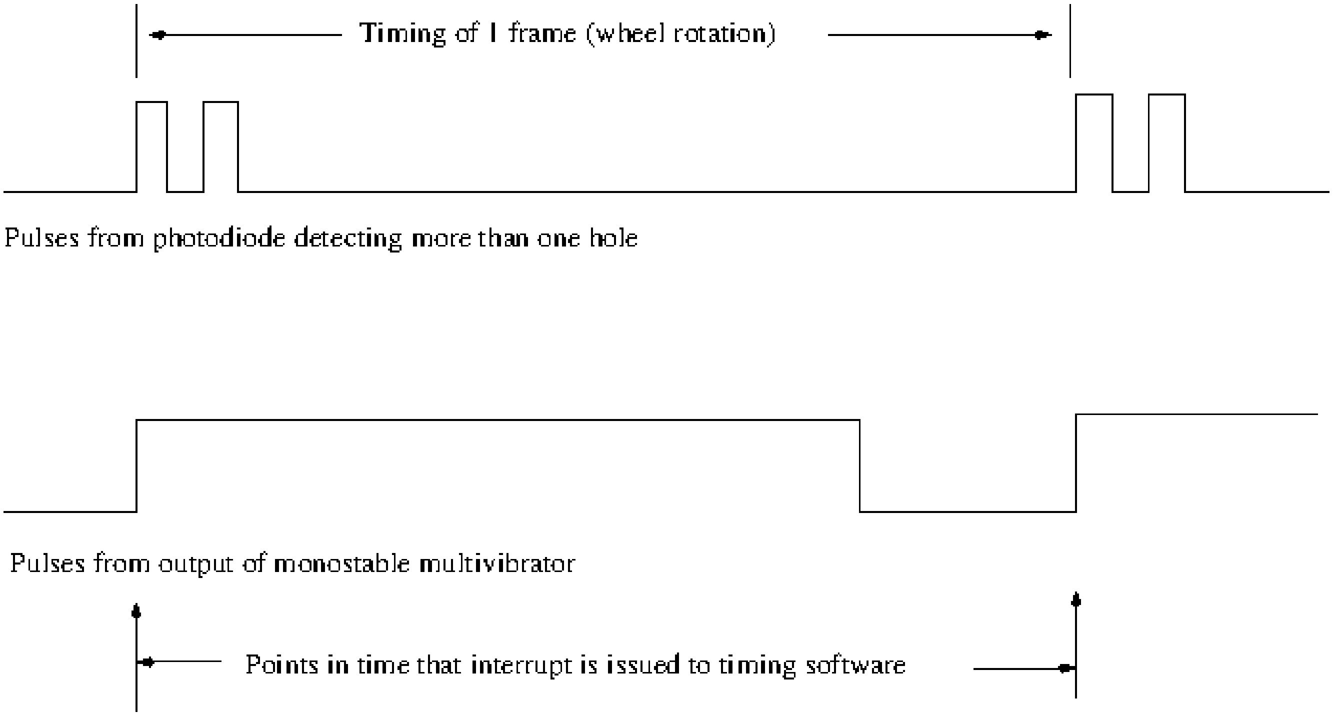


This next figure shows where everything is when the Nipkow disk rotates by one hole:



Please note that the difference in radius from the axis (center) of the disk is smaller than the aparture of the PhotoDiode. In reality, more than two holes can flash by and be 'seen' by the PhotoDiode. We use two to simplify what is happening.

The following figure shows the relationship between the two pulese from the two holes that are 'seen' by the PhotoDiode and the output of the monostable multivibrator:

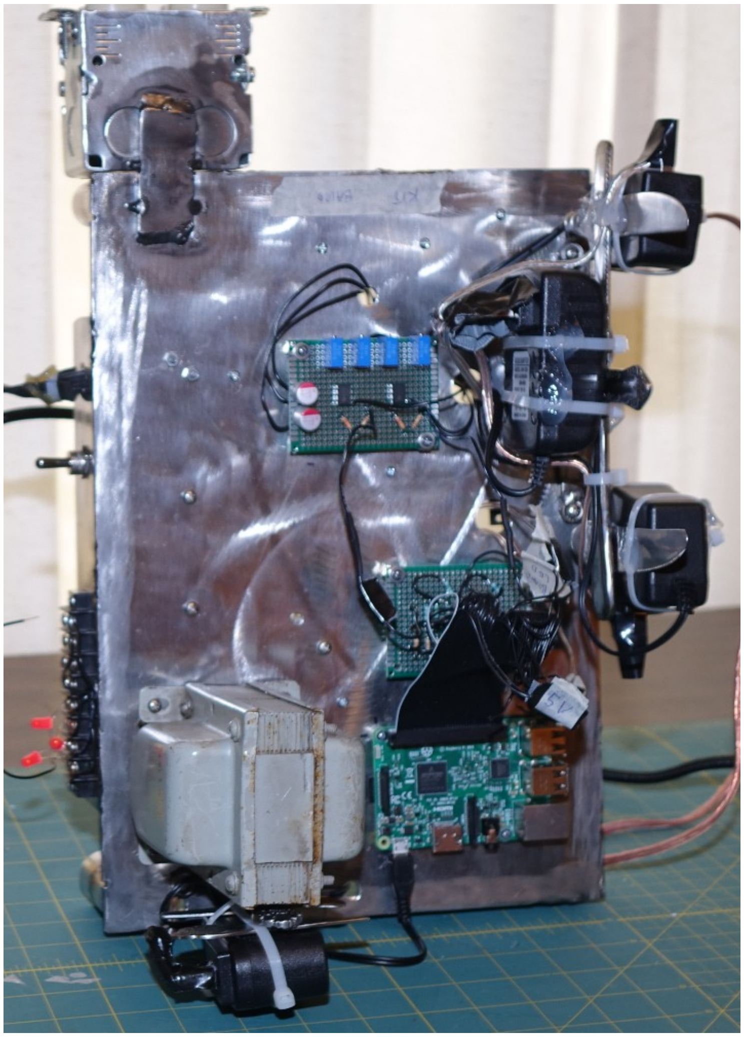

Timing of I frame (wheel rotation)
Pulses from photodiode detecting more than one hole
Pulses from output of monostable multivibrator
Points in time that interrupt is issued to timing software

The duration of the monostable multivibrator is set to be long enough to cover the timing of about 7 holes. We estimate that about four holes are 'seen' due to the size of the aparture of the PhotoDiode.

There is one more very important thing to note here. The output of the LM555 multivibrator is fed into a resistor divider. This is important because the inputs of the RaspBerry PI are limited to 3 volts and the LM555 has a 5 volt TTL logical output. Putting 5 volts into a GPIO input can damage the RaspBerry PI.

Photographs showing component locations

Top of chassis

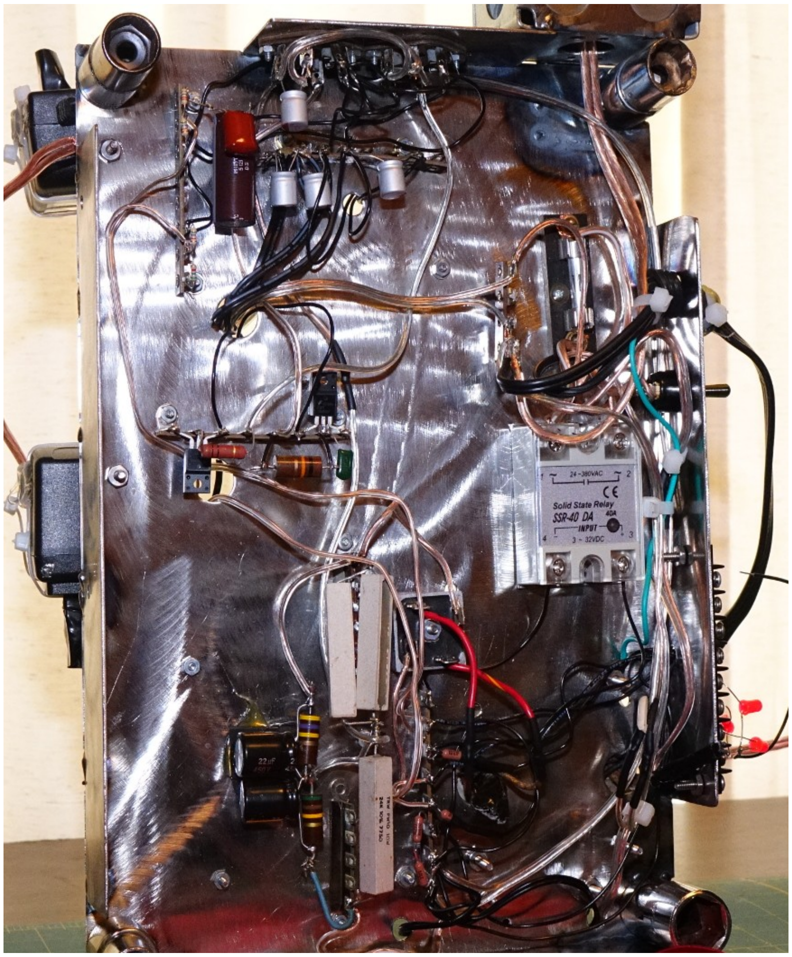


Looking at the top of the chassis with the large power transformer on the lower left hand corner of the image:

1. The 5 volt power supply is visible beneath the large power transformer
2. The large power transformer itself is part of the +300 volt power supply for the neon bulb.

3. The Raspberry PI is located to the right of the larage power transformer
4. The Digital to Analog converter (the resistor ladder) is located adjacent to the Raspberry PI. It is obscured by the ribbon cable plugged into the Raspberry PI
5. Above the Digital to Analong converter is the Analog Signal OP Amp. It is the circuit board with the four blue potentiometers.
6. To the right of the Analog Signal OP Amp circuit board is the hand made metal frame that is holding the +12, -12, and +15 volt power supplies.
7. The box that extends from the end of the chassis (located on the upper left hand corner of the picture) is the ac power outlet for the mechanical television motors.

Bottom of chassis

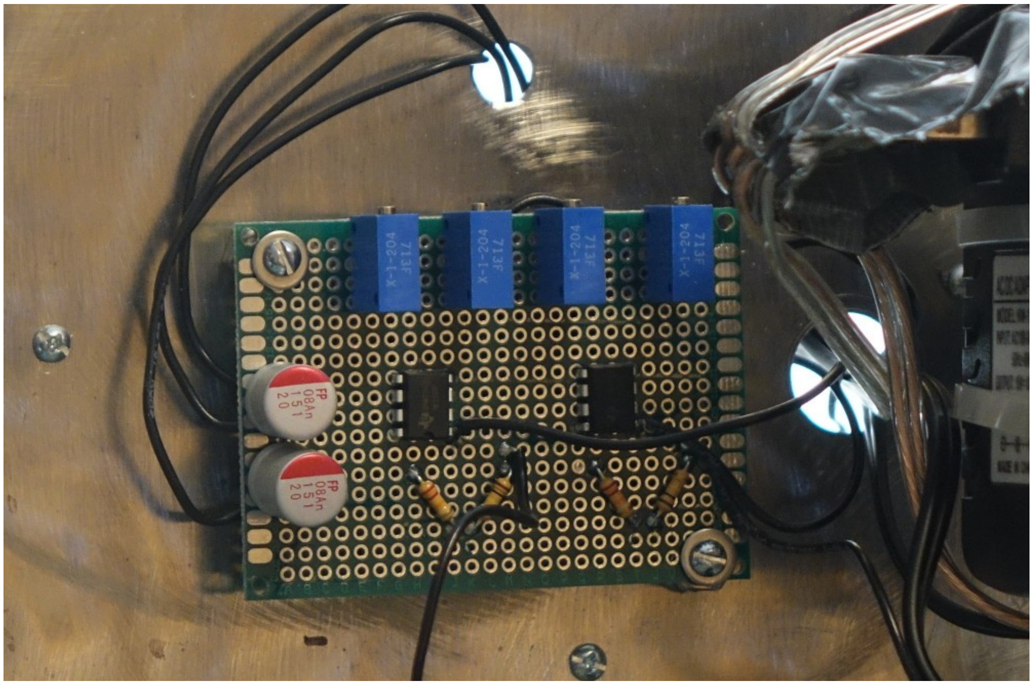


Looking at the bottom of the chassis, starting from the upper left hand corner of the image above:

1. The terminal strip on the extreme upper left hand corner of the image (adjacent to the chassis leg) is where the timing signals arrive from the timing circuits (the LED and PhotoDiode circuits) in each of the mechanical televisions. This is where the resistor devider is to reduce the 5 volt TTL level to the 3 volt logic level required by the RaspBerry PI.
2. Adjacent the terminal strip (on the right) are some filter capacitors for the +5, -12, +12, and +15 volt power supplies. At this point, the different power supplies are furnished to the octal connectors (they are located on the edge of the chassis at the top center of the photograph).
3. To the right of the filter capacitors is the AC line power handling components consisting of the power cord, the fuse, and the switch. Just below that is the solid state relay the controls the AC power to the outlets and the -12, +12, +15, and +300 volt power supplies. It is the gray unit with the four heavy screw connectors.

4. To the left of the solid state relay, you see a terminal strip with what appear to be two large resistors. If you look carefully, you will see two MOSFETs. Those are the high voltage and high current MOSFETs; one for the neon bulb and one for the LED array. The two resistors are the current limiting resistors that are connected to the drain pin of each of the MOSFETs.
5. The bottom half of the photograph is the circuitry for the 300 volt power supply for the neon bulb. The bridge rectifier block is the small square with the four pins that is between the solid state relay and the pair of 5 watt resistors.
6. The three 5 watt resistors are for bleeding the power supply and the filter capacitors are located just to the left of the 5 watt resistors.

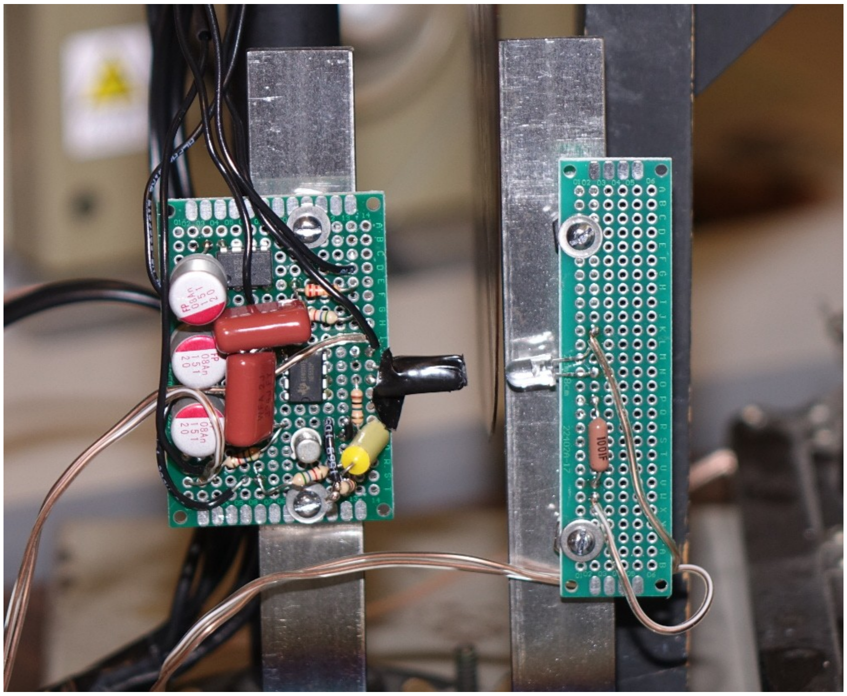
Signal processor board



There are four potentiometers on the signal processor board. In order starting from the left (where the two electrolytic capacitors are:

1. Baird offset
2. Baird gain
3. Kit offset
4. Kit gain

Optical sensor board

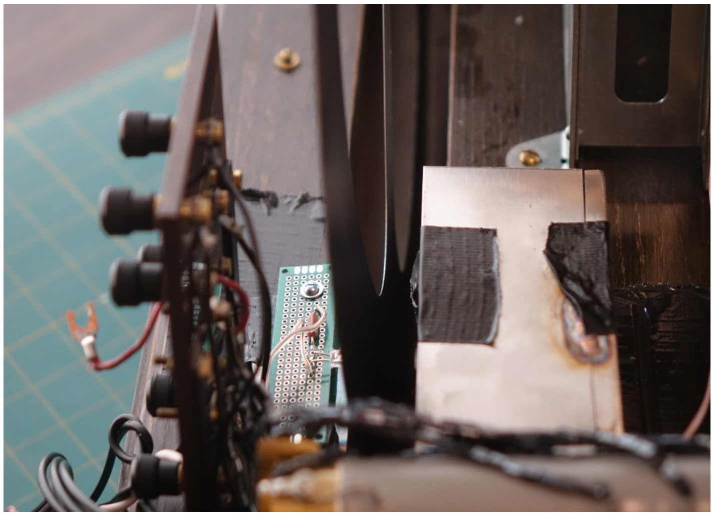


This is the optical sensor assembly for generating the timing signals from the kit mechanical television. The Nipkow disk is between the two printed circuit boards; it is difficult to see because it is viewed from its edge.

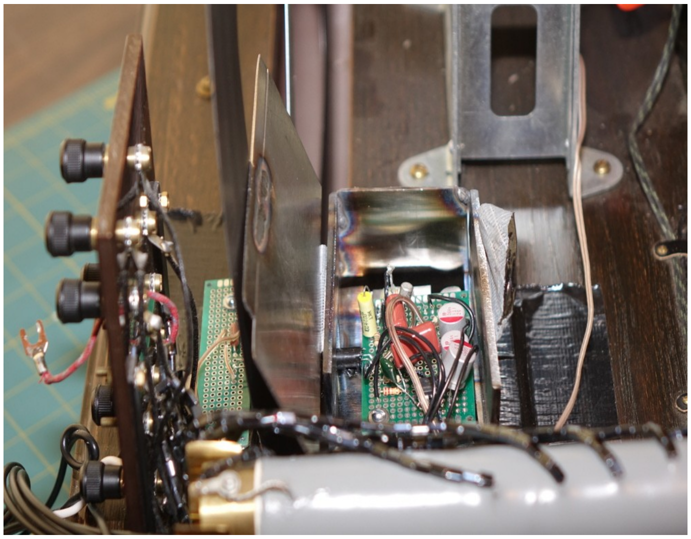
The circuit board on the right is for the infrared LED. The circuit board on the left is for the infrared PhotoDiode. The PhotoDiode is within the black cylinder that extends to the right. The IC chip adjacent to the PhotoDiode is the OP Amp. The 2N2222 transistor is below the OP Amp. The IC chip toward the upper left hand corner of the photo is the LM555 timer set up as a monostable multivibrator.

Please note that no modifications were made to the kit mechanical television. The mounting of the sensor assembly is done using an existing bolt that holds down the magnifying glass.

Optical sensor with shield for Baird



Due to excessive electrical noise from the motor of the Baird, the optical sensor circuit board has to be shielded. In the above photo, the infrared LED assembly is to the left and the metal shield with the duct tape is to the right. The Nipkow disk is in the center (hard to see due to being viewed from the edge).



Here is the optical sensor circuit board with the shield open.

Please note that the optical sensor is glued using as soft silicone glue as well as duct taped to the Baird. There have been no permanent modifications made to the Baird; the sensor assembly can be removed and the Baird will be in it's original condition.

Pin tables

Ribbon Cable for RaspBerry PI

Following are the pins for the ribbon cable that is attached to the RaspBerry PI. Numbering is as viewed from the top of the RaspBerry PI with the in array on the right hand side:

3 volts	5 volts
Baird video bit 0	5 volts
Baird video bit 1	Ground
Baird video bit 2	Baird video bit 3
Ground	Kit video bit 2
Kit video bit 0	Kit video bit 3
Kit video bit 1	Ground
Baird timing	Kit timing
3 volts	Activation pushbutton
Solid State Relay Input	Ground
Baird Manual Sync	Baird Manual Sync LED
Pins beyond this point are not used	

Octal socket (both sockets pinned the same)

1	Ground
2	+ 5 Volts (used for timing)
3	Timing Output
4	- 12 Volts (used for timing)
5	+ 12 Volts (used for timing)
6	Light source puldown
7	Light source power
8	Not Used

Please note that pin 6 is the pin that is connected to the MOSFET (through a current liminting resistor) the controls the current through the light source (neon bulb for the kit and LED array for the Baird). Pin 7 is for light source power (+15 volts for the LED array and +300 volts for the neon bulb).

Software code listing

The software was written in the C programming language and is running at all times on the Raspberry PI. The RaspBerry PI runs a Linux operating system that boots up when the RaspBerry PI is powered on. Upon boot-up, the Linux operating system starts the software for the image converter.

Please note that some of the code is used for testing and is not used for normal operation. I chose to leave that code in place so that others can use it if the need arises.

```
/* Copyright (c) 2017 Mark Allyn, Bellingham, Washington. */
/* Distributed under BSD License */

#include <pthread.h>
#include <stdlib.h>
#include <stdio.h>
#include <errno.h>
#include <time.h>
#include <sched.h>

#include <X11/Xlib.h>
#include <X11/Xutil.h>
#include <X11/Xos.h>
#include <assert.h>
#include <unistd.h>

#include <gif_lib.h>
#include <wiringPi.h>

#define BILLION 1000000000
#define BAIRD_ROWS 30
#define BAIRD_COL 40
#define KIT_ROWS 48
#define KIT_COL 28
#define BAIRD_DELAY 13
#define KIT_DELAY 60

#define BITB0 2
#define BITB1 3
#define BITB2 4
#define BITB3 14

#define BITA0 17
#define BITA1 27
#define BITA2 15
#define BITA3 18
```

```
#define BAIRD 22
#define KIT 23

#define RELAY 10
#define BUTTON 24

#define BRAKE_BUT 9
#define BRAKE 25

static char *filename;

static unsigned char *baird_buf = NULL;
static unsigned char *kit_buf = NULL;

static GifRecordType RecordType;
static GifByteType *Extension = NULL;
static GifFileType *GifFile = NULL;

static int Error = 0;
static int brake_status = 0;
static int brake_use = 0;
static int brake_cycle_count = 0;

static int kit_frame_count;
static int baird_frame_count;
static struct timespec baird_duration;
static struct timespec kit_duration;
static long baird_delay;
static int kit_delay;
static struct timespec fix_timer_interval;
static struct timespec baird_frame_start;
static struct timespec kit_frame_start;
static struct timespec baird_delay_time;
static struct timespec kit_delay_time;
static struct timespec baird_frame_end;
static struct timespec kit_frame_end;
static struct timespec button_time;

static int InterlacedOffset[] = { 0, 4, 2, 1 };
static int InterlacedJumps[] = { 8, 8, 4, 2 };
static int ImageNum = 0;
static ColorMapObject *ColorMap = NULL;

static GifWord screen_width = 0;
static GifWord screen_height = 0;
static int image_count = 0;
static size_t movie_pixel_count = 0;
static size_t image_pixel_count = 0;

static pthread_t fix_timer_pthread;
static pthread_t delay_entry_pthread;
static pthread_t baird_timer_pthread;
static pthread_t kit_timer_pthread;
static pthread_t scan_baird_pthread;
static pthread_t scan_kit_pthread;
static pthread_t button_pthread;
static pthread_mutex_t baird_lock;
static pthread_mutex_t kit_lock;

static int stop_it = 0;

/* used for testing */
struct display_struct {
  Display *my_display;
  int blackcolor;
  int whitecolor;
  Window my_window;
  Window my_window_orig;
  GC my_gc;
  GC my_gc_orig;
  XEvent my_event;
  XImage *my_image;
  Visual *my_visual;
  Colormap my_colormap;
};

typedef struct display_struct display_struct_t;

static display_struct_t my_ds;

/* used for testing */
void open_display(display_struct_t *ds)
{

  ds->my_display = XOpenDisplay(0);
  if (ds->my_display == NULL) {
    perror("cant open display");
    exit(EXIT_FAILURE);
  }

  assert(ds->my_display);

  ds->blackcolor = BlackPixel(ds->my_display, DefaultScreen(ds-
>my_display));
  ds->whitecolor = WhitePixel(ds->my_display, DefaultScreen(ds-
>my_display));

  ds->my_visual = DefaultVisual(ds->my_display, 0);

  ds->my_window = XCreateSimpleWindow(ds->my_display,
DefaultRootWindow(ds->my_display),
    0, 0, BAIRD_ROWS*7, BAIRD_COL*7, 0, ds->blackcolor, ds-
>blackcolor);

  ds->my_window_orig = XCreateSimpleWindow(ds->my_display,
DefaultRootWindow(ds->my_display),
    300, 300, 900, 900, 0, ds->blackcolor, ds->blackcolor);

  XSelectInput(ds->my_display, ds->my_window, StructureNotifyMask);
  XSelectInput(ds->my_display, ds->my_window_orig,
StructureNotifyMask);
  XMapWindow(ds->my_display, ds->my_window);
  XMapWindow(ds->my_display, ds->my_window_orig);
  ds->my_gc = XCreateGC(ds->my_display, ds->my_window, 0, 0);
  ds->my_gc_orig = XCreateGC(ds->my_display, ds->my_window_orig, 0,
0);
  XSetForeground(ds->my_display, ds->my_gc, ds->whitecolor);
  XSetBackground(ds->my_display, ds->my_gc, ds->blackcolor);
  XSetFillStyle(ds->my_display, ds->my_gc, FillSolid);
  XSetForeground(ds->my_display, ds->my_gc_orig, ds->whitecolor);
  XSetBackground(ds->my_display, ds->my_gc_orig, ds->blackcolor);
  XSetFillStyle(ds->my_display, ds->my_gc_orig, FillSolid);
  ds->my_colormap = DefaultColormap(ds->my_display, DefaultScreen(ds-
>my_display));

  XFlush(ds->my_display);

  for(;;) {
    XNextEvent(ds->my_display, &ds->my_event);
    printf("event %d\n", ds->my_event.type);
    if (ds->my_event.type == MapNotify)
      break;
  }
}

/* used for testing */
void display_orig(display_struct_t *ds, struct SavedImage *image)
{
  int x_count, y_count, pixel_count;
  unsigned char color_map_index;
  XColor my_color;
  int orig_color;
  pixel_count = 0;
  for (y_count = 0; y_count < screen_height; y_count += 1) {
    for (x_count = 0; x_count < screen_width; x_count += 1) {
      color_map_index = image->RasterBits[pixel_count];
      orig_color = ColorMap->Colors[color_map_index].Red;
      //printf("Colormap index is %d and red is %d\n",
color_map_index, orig_color);
      //printf("xcount is %d ycount %d\n", x_count, y_count);
      pixel_count += 1;
      my_color.red = orig_color * 200;
      my_color.green = orig_color * 200;
      my_color.blue = orig_color * 200;
      my_color.flags = DoRed | DoBlue | DoGreen;
      XAllocColor(ds->my_display, ds->my_colormap, &my_color);
      XSetForeground(ds->my_display, ds->my_gc_orig, my_color.pixel);
      XFillRectangle(ds->my_display, ds->my_window_orig, ds-
>my_gc_orig,
        x_count, y_count, 1, 1);
    }
  }
}

/* used for testing */
void do_pixel(display_struct_t *ds, int x, int y, int val)
{
  XColor my_color;
  my_color.red = (65000 / 16) * val;
  my_color.blue = (65000 / 16) * val;
  my_color.green = (65000 / 16) * val;
  my_color.flags = DoRed | DoBlue | DoGreen;
  XAllocColor(ds->my_display, ds->my_colormap, &my_color);
  XSetForeground(ds->my_display, ds->my_gc, my_color.pixel);
  XFillRectangle(ds->my_display, ds->my_window, ds->my_gc,
    x * 3, y * 3, 3, 3);
  XFlush(ds->my_display);
}

/* Convert cluster of input pixels to a single output pixel */
int get_converted_pixel(struct SavedImage *image, int out_width,
  int out_height, int out_x, int out_y)
{
  int box_width;
  int box_height;
  int box_x;
  int box_y;
  int cur_x;
  int cur_y;
  int sum;
  unsigned char color_map_index;
  /* horiz size of pixel box */
  box_width = screen_width / out_width;
  box_height = screen_height / out_height;
  box_x = out_x * box_width;
  box_y = out_y * box_height;
  sum = 0;
  for (cur_y = 0; cur_y < box_height; cur_y += 1) {
    for (cur_x = 0; cur_x < box_width; cur_x += 1) {
      color_map_index = image->RasterBits[(box_x + cur_x) + ((box_y +
cur_y) * screen_width)];
      sum += ColorMap->Colors[color_map_index].Red;
    }
  }
  return (sum / (box_width * box_height));
}

/* We are outputting only four bits to digital to anolog converter */
unsigned char fourbit(unsigned char eightbit)
{
  unsigned char fourbitsum;
  fourbitsum = 0;
  fourbitsum = eightbit/16;
//  if ((0b00000011) & eightbit) fourbitsum |= 0b00000001;
//  if ((0b00001100) & eightbit) fourbitsum |= 0b00000010;
//  if ((0b00110000) & eightbit) fourbitsum |= 0b00000100;
//  if ((0b11000000) & eightbit) fourbitsum |= 0b00001000;
  return fourbitsum;
}

/* Convert one frame */
void process_image(int output_width, int output_height,
  struct SavedImage *image, unsigned char *output_buf)
{
  unsigned char *cptr1;
  int working;
  int x_count, y_count, pixel_count;
  x_count = 0;
  y_count = 0;
  pixel_count = 0;
  cptr1 = output_buf;
  for (y_count = 0; y_count < output_height; y_count += 1) {
    for (x_count = 0; x_count < output_width; x_count += 1) {
      working = get_converted_pixel(image, output_width,
output_height,
         x_count, y_count);
      *cptr1 = fourbit(working);
      //printf("x_count %d y_count %d working %d fourbit %d\n",
      //  x_count, y_count, working, *cptr1);
      cptr1 += 1;
    }
  }
}

/* convert an entire movie */
void process_movie(int output_width, int output_height, unsigned char
*movie_buf)
{
  int frame_count;
  unsigned char *image_buf;
  image_buf = movie_buf;
  for (frame_count = 0; frame_count < image_count; frame_count += 1)
  {
    process_image(output_width, output_height,
      GifFile->SavedImages + frame_count, image_buf);
    image_buf += (output_width * output_height);
    //printf("Image was %d of %d screen width %d screen height %d
output_width %d, output_height %d\n",
    //  frame_count, image_count, screen_width, screen_height,
output_width, output_height);
  }
}

/* convert both of our movies */
void process_both_movies(void)
{
  /* Note for baird, col and row are reversed as scans are vertical
*/
  process_movie(BAIRD_ROWS, BAIRD_COL, baird_buf);
  process_movie(KIT_COL, KIT_ROWS, kit_buf);
}

void outputa(char val)
{
  /* printf("%d %d %d %d\n", val & 0x01, val & 0x02, val & 0x04, val
& 0x08); */
  if ((val & 0x01) == 0) digitalWrite(BITA0, HIGH);
  if ((val & 0x01) != 0) digitalWrite(BITA0, LOW);

  if ((val & 0x02) == 0) digitalWrite(BITA1, HIGH);
  if ((val & 0x02) != 0) digitalWrite(BITA1, LOW);
  if ((val & 0x04) == 0) digitalWrite(BITA2, HIGH);
  if ((val & 0x04) != 0) digitalWrite(BITA2, LOW);

  if ((val & 0x08) == 0) digitalWrite(BITA3, HIGH);
  if ((val & 0x08) != 0) digitalWrite(BITA3, LOW);
}

void outputb(char val)
{
  /* printf("%d %d %d %d\n", val & 0x01, val & 0x02, val & 0x04, val
& 0x08); */
  if ((val & 0x01) == 0) digitalWrite(BITB0, HIGH);
  if ((val & 0x01) != 0) digitalWrite(BITB0, LOW);

  if ((val & 0x02) == 0) digitalWrite(BITB1, HIGH);
  if ((val & 0x02) != 0) digitalWrite(BITB1, LOW);

  if ((val & 0x04) == 0) digitalWrite(BITB2, HIGH);
  if ((val & 0x04) != 0) digitalWrite(BITB2, LOW);

  if ((val & 0x08) == 0) digitalWrite(BITB3, HIGH);
  if ((val & 0x08) != 0) digitalWrite(BITB3, LOW);
}

void setupgif(void)
{
  /* Open gif file and do slurp read */
  printf("Opening %s\n", filename);
  GifFile = DGifOpenFileName(filename);
  if (GifFile == NULL) {
    fprintf(stderr, "cannot open gif file\n");
    perror("Error opening gif file");
    exit(EXIT_FAILURE);
  }
  Error = DGifSlurp(GifFile);
  if (Error != GIF_OK) {
    fprintf(stderr, "slurp returns %d\n", Error);
    PrintGifError();
    exit(EXIT_FAILURE);
  }

  screen_width = GifFile->SWidth;
  screen_height = GifFile->SHeight;
  image_count = GifFile->ImageCount;
  ColorMap = GifFile->SColorMap;
  /* Allocate the baird and kit video buffers */
  /* These are monochrome and much smaller than */
  /* original gif; so we will open entire movie */
  /* for each and then do full conversion prior */
  /* to playback to save performance */
  baird_buf = calloc(BAIRD_ROWS * BAIRD_COL * image_count, 1);
  if (baird_buf == NULL) {
    fprintf(stderr, "cannot open baird buffer\n");
    perror("Error opening baird buffer");
    exit(EXIT_FAILURE);
  }
  kit_buf = calloc(KIT_ROWS * KIT_COL * image_count, 1);
  if (kit_buf == NULL) {
    fprintf(stderr, "cannot open kit buffer\n");
    perror("Error opening kit buffer");
    exit(EXIT_FAILURE);
  }
}

/* time functions */
/* Add timespec */
struct timespec time_add(struct timespec time1, struct timespec
time2)
{
  struct timespec working;
  working.tv_nsec = (long)0;
  working.tv_sec = (time_t)0;
  if (time1.tv_nsec + time2.tv_nsec >= (long)BILLION) {
    working.tv_sec += 1;
    working.tv_sec += time1.tv_sec + time2.tv_sec;
    working.tv_nsec = time1.tv_nsec + time2.tv_nsec - (long)BILLION;
  } else {
    working.tv_sec = time1.tv_sec + time2.tv_sec;
    working.tv_nsec = time1.tv_nsec + time2.tv_nsec;
  }
  return working;
}

/* Subtract timespec (result must be positive) */
/* Value into *time_result; returns 1 for positive */
/* or -1 for negative */
int time_sub(struct timespec *time_result, struct timespec time1,
  struct timespec time2)
{
  /* time1 - time2 */
  int result;
  struct timespec working;
  working.tv_nsec = (long)0;
  working.tv_sec = (time_t)0;

  if ((time1.tv_nsec - time2.tv_nsec) < 0) {
    working.tv_sec = time1.tv_sec - time2.tv_sec - 1;
    working.tv_nsec = time1.tv_nsec - time2.tv_nsec + (long)BILLION;
  } else {
    working.tv_sec = time1.tv_sec - time2.tv_sec;
    working.tv_nsec = time1.tv_nsec - time2.tv_nsec;
  }

  if (working.tv_sec < 0) result = -1;
  else result = 1;
  *time_result = working;
  return result;
}

/* Return time in nanoseconds (risky if over 1 second duration) */
long nsec_time(struct timespec input_time)
  {
  long working;
  working = (long)0;
  if (input_time.tv_sec > 0) {
    working += (long)BILLION;
  }
  working += input_time.tv_nsec;
  return working;
}

/* Division; this only works for up to 2 seconds on the total time */
struct timespec devide_time(struct timespec total_time, int devider)
{
  struct timespec working;
  int ct1;
  working.tv_sec = total_time.tv_sec;
  working.tv_nsec = total_time.tv_nsec;
  if (working.tv_sec > 2) working.tv_sec = 2;
  working.tv_nsec += (working.tv_sec * (long)BILLION);
  working.tv_nsec = working.tv_nsec / (long)devider;
  for(working.tv_sec = 0; working.tv_nsec >= (long)BILLION;
working.tv_sec += 1) {
    working.tv_nsec -= (long)BILLION;
  }
  return working;
}

void *fixtimer(void *nothing)
```

```
{
  static struct timespec start_time;
  static struct timespec stop_time;
  static long stop_nsec;
  static long start_nsec;
  static long duration;
  /* avg_duration needs to be long long in case totaling for average
*/
  /* results in value over 4 billion (max for long) */
  static long long avg_duration_total;
  static struct timespec duration_timespec;
  static struct timespec working_baird_duration;
  static struct timespec working_baird_delay;
  static struct timespec working_baird_frame_start;
  static struct timespec working_baird_frame_end;
  static struct timespec diff_timespec;
  static long avg_bucket[20];
  static int ct1;
  static int baird_working_frame_count;
  static int last_state;
  static int my_error;
  static int my_direction;

  for (ct1 = 0; ct1 < 20; ct1 += 1) avg_bucket[ct1]  = (long)0;
  last_state = 0;

  /* initial grab of times */
  my_error = clock_gettime(CLOCK_REALTIME, &stop_time);
  if (my_error) {
    perror("clock_gettime");
    exit(1);
  }

  while(1) {

    if (brake_use == 0) {
      /* not being used if normal timing */
      sched_yield();
      continue;
    }

    do {
      my_error = clock_gettime(CLOCK_REALTIME, &start_time);
      if (my_error) {
        perror("clock_gettime");
        exit(1);
      }
      my_direction = time_sub(&diff_timespec, start_time, stop_time);
      if (my_direction < 0) {
        sched_yield();
        continue;
      }
    } while (my_direction < 0);

    /* We reached stop time */
    /* Now bump up stop time */
    stop_time = time_add(start_time, fix_timer_interval);

    working_baird_duration.tv_sec = fix_timer_interval.tv_sec;
    working_baird_duration.tv_nsec = fix_timer_interval.tv_nsec;

    /* Calculate next frame start; needs to use delay, which is */
    /* percentage of the total duration time */
    working_baird_delay.tv_sec = (time_t)0;
    working_baird_delay.tv_nsec = working_baird_duration.tv_nsec /
(long)100;
    working_baird_delay.tv_nsec = working_baird_delay.tv_nsec *
baird_delay;
    working_baird_frame_start = time_add(start_time,
      working_baird_delay);
    working_baird_frame_end = time_add(working_baird_frame_start,
      working_baird_duration);

    /* now copy in after getting lock */
    pthread_mutex_lock(&baird_lock);
    baird_frame_count += 1;
    baird_duration = working_baird_duration;
    baird_frame_start = working_baird_frame_start;
    baird_frame_end = working_baird_frame_end;
    baird_delay_time = working_baird_delay;
    pthread_mutex_unlock(&baird_lock);
    // printf("fix duration %ld sec %ld nsec\n",
baird_duration.tv_sec, baird_duration.tv_nsec);

  }
  pthread_exit(NULL);
}

void bairdtimer(void)
{
  static struct timespec start_time;
  static struct timespec stop_time;
  static long stop_nsec;
  static long start_nsec;
  static long duration;
  /* avg_duration needs to be long long in case totaling for average
*/
  /* results in value over 4 billion (max for long) */
  static long long avg_duration_total;
  static struct timespec duration_timespec;
  static struct timespec working_baird_duration;
  static struct timespec working_baird_delay;
  static struct timespec working_baird_frame_start;
  static struct timespec working_baird_frame_end;
  static long avg_bucket[20];
  static int ct1;
  static int baird_working_frame_count;
  static int last_state;
  static int my_error;
  static int my_direction;

  if (brake_use == 1) return;

  my_error = clock_gettime(CLOCK_REALTIME, &stop_time);
  if (my_error) {
    perror("clock_gettime");
    exit(1);
  }

  my_direction = time_sub(&duration_timespec, stop_time, start_time);

  /* Corner case if machine first start or glitch */
  if ((my_direction < 0) || (duration_timespec.tv_sec > 2)) {
    duration = (long)1000000;
  } else {
    duration = nsec_time(duration_timespec);
  }

  working_baird_duration.tv_sec = 0;
  working_baird_duration.tv_nsec = (long)duration;

  /* Calculate next frame start; needs to use delay, which is */
  /* percentage of the total duration time */
  working_baird_delay.tv_sec = (time_t)0;
  working_baird_delay.tv_nsec = working_baird_duration.tv_nsec /
(long)100;
  working_baird_delay.tv_nsec = working_baird_delay.tv_nsec *
baird_delay;
  working_baird_frame_start = time_add(stop_time,
    working_baird_delay);
  working_baird_frame_end = time_add(working_baird_frame_start,
    working_baird_duration);

  /* now copy in after getting lock */
  baird_frame_count += 1;
  baird_duration = working_baird_duration;
  baird_frame_start = working_baird_frame_start;
  baird_frame_end = working_baird_frame_end;
  baird_delay_time = working_baird_delay;

  /* New Rotation Starting, Move Stop to Start */
  start_time = stop_time;
}

void *scan_baird(void *nothing)
{
  static int my_x, my_y, my_image;
  static unsigned char *bufptr;
  static unsigned char *work_pointer;
  static unsigned char testchar;
  static struct timespec my_baird_frame_duration;
  static struct timespec my_baird_frame_start;
  static struct timespec my_baird_frame_end;
  static struct timespec my_baird_delay_time;
  static struct timespec working_baird_frame_start;
  static struct timespec baird_line_start;
  static struct timespec baird_line_duration;
  static struct timespec baird_pixel_start;
  static struct timespec baird_pixel_duration;
  static struct timespec current_time;
  static struct timespec time_difference;
  static struct timespec time_sum;
  static int time_direction;
  static int horizontal_offset;
  static int work_y, work_x;
  static int my_baird_frame_count;
  static int working_baird_frame_count;
  static int image_rows;
  static int image_col;
  static int this_image_row;
  static int this_image_col;

  /* Initialize to 0 */
  working_baird_frame_start.tv_sec = 0;
  working_baird_frame_start.tv_nsec = (long)0;

  image_rows = BAIRD_COL;
  image_col = BAIRD_ROWS;

  // /* TESTING */
  // printf("image_count %d image rows %d image columns %d\n",
image_count,
  //   BAIRD_COL, BAIRD_ROWS);
  // bufptr = baird_buf;
  // if (baird_frame_count < 3) {
  //   for(work_y = 0; work_y < image_rows; work_y += 1) {
  //     int my_test_count;
  //     for (work_x = 0; work_x < image_col; work_x += 1) {
  //       *bufptr = (unsigned char)0;
  //     if ((work_x == 5) && (work_y >= 5) && (work_y < image_rows
-5)) *bufptr = (unsigned char)10;
  //     if ((work_x == image_col -5) && (work_y >= 5) && (work_y <
image_rows -5)) *bufptr = (unsigned char)10;
  //     if ((work_y == 5) && (work_x >= 5) && (work_x < image_col
-5)) *bufptr = (unsigned char)10;
  //     if ((work_y == image_rows -5) && (work_x >= 5) && (work_x <
image_col -5)) *bufptr = (unsigned char)10;
  //     printf("val %d for x %d of %d and y %d of %d\n",
  //       (unsigned char)*bufptr, work_x, image_col, work_y,
image_rows);
  //     bufptr += 1;
  //     }
  //   }
  // }

  while(1) {
    /* Continually loop on film */
    bufptr = baird_buf;
    for (my_image = 0; my_image < image_count; my_image += 1) {
      /* on each frame */
      /* set pixel buffer to start of the new frame */
      this_image_row = 0;
      this_image_col = 0;
      bufptr = baird_buf + (BAIRD_ROWS * BAIRD_COL * my_image);

      /* Now wait for new frame start */
      /* new frame starts when my frame start time is no longer */
      /* equel to system frame start time */
      do {
        /* first wait for timer frame count to increment */
        pthread_mutex_lock(&baird_lock);
        working_baird_frame_count = baird_frame_count;
        pthread_mutex_unlock(&baird_lock);
        if (working_baird_frame_count == my_baird_frame_count) {
          /* timer's baird frame count has not advanced yet */
          sched_yield();
        }
      } while (working_baird_frame_count == my_baird_frame_count);
      /* Now we have new frame count; we still must wait for
         new frame start as we still may be in the delay */
      do {
        pthread_mutex_lock(&baird_lock);
        working_baird_frame_start = baird_frame_start;
        pthread_mutex_unlock(&baird_lock);

        clock_gettime(CLOCK_REALTIME, &current_time);

        time_direction = time_sub(&time_difference,
          current_time, working_baird_frame_start);
        if (time_direction < 0) {
          /* We are still before the new frame's start */
          sched_yield();
        }
      } while (time_direction < 0);

      /* Now at new frame start */
      /* set new times */
      pthread_mutex_lock(&baird_lock);
      my_baird_frame_count = baird_frame_count;
      my_baird_frame_duration = baird_duration;
      my_baird_frame_start = baird_frame_start;
      my_baird_frame_end = baird_frame_end;
      my_baird_delay_time = baird_delay_time;
      pthread_mutex_unlock(&baird_lock);
      baird_line_start = my_baird_frame_start;
      baird_pixel_start = my_baird_frame_start;
      /* Baird lines are vertical */
      /* Scan is from upper right to lower left, spot going down for
each row
         and moving left for each column. Paint vertical rows from
right
         to left; moving spot from top to bottom */
      baird_line_duration = devide_time(my_baird_frame_duration,
BAIRD_ROWS);
      baird_pixel_duration = devide_time(baird_line_duration,
BAIRD_COL);
      // printf("frame: %ld line: %ld pixel: %ld\n",
        // my_baird_frame_duration.tv_nsec,
baird_line_duration.tv_nsec,
        // baird_pixel_duration.tv_nsec);
      /* Now wait for the start of this frame in real time*/
      do {
        clock_gettime(CLOCK_REALTIME, &current_time);
        time_direction = time_sub(&time_difference,
          current_time, my_baird_frame_start);
      } while (time_direction < 0);

      // printf("Processing Frame\n");

      /* Baird Rows are vertical; columns are horizontal */
      /* Now the housekeeping is done for the frame; now output it by
line */
      for (my_x = 0; my_x < BAIRD_ROWS; my_x += 1) {
        /* Lines are vertical; starting from top right */
        horizontal_offset = (BAIRD_COL - 1) - my_x;
        /* Lets make sure that we have not run out of time for the
frame */
        pthread_mutex_lock(&baird_lock);
        working_baird_frame_start = baird_frame_start;
        pthread_mutex_unlock(&baird_lock);
        time_direction = time_sub(&time_difference,
my_baird_frame_start,
          working_baird_frame_start);
        if ((time_difference.tv_sec != 0) ||
          (time_difference.tv_nsec != 0)) {
          /* System has new time - lets make sure we are still within
the
             time allocated for this current frame. Grab the system
time;
             we still may be in the delay window */
          clock_gettime(CLOCK_REALTIME, &current_time);
          time_direction = time_sub(&time_difference,
working_baird_frame_start,
            current_time);
          if (time_direction < 0) {
            /* We've shot beyond the time for the new frame; bail out
*/
            // printf("baird turnover at %d\n", my_y);
            break;
          }
        }
        /* now wait for this line's start time */
        do {
          clock_gettime(CLOCK_REALTIME, &current_time);
          time_direction = time_sub(&time_difference,
            current_time, baird_line_start);
        } while (time_direction < 0);
        // printf("line\n");
        /* now output the individual pixels for the line */
        for (my_y = 0; my_y < BAIRD_COL; my_y += 1) {
          /* work values to point to pixel in memory, which */
          /* is by horizontal line for GIF files */
          work_y = (BAIRD_COL -1 - my_y) * BAIRD_ROWS;
          work_x = horizontal_offset;
          work_pointer = bufptr;
          work_pointer = work_pointer + work_y + work_x;
          /* Wait for the pixel start */
          clock_gettime(CLOCK_REALTIME, &current_time);
          time_direction = time_sub(&time_difference,
            current_time, baird_pixel_start);
          if (time_direction < 0) {
            do {
              clock_gettime(CLOCK_REALTIME, &current_time);
              // printf("pw\n");
              time_direction = time_sub(&time_difference,
                current_time, baird_pixel_start);
            } while (time_direction < 0);
          }
          /* twiddle the brake pulse if appropriate */
          if (brake_use) {
            if (my_y < (BAIRD_COL / 2)) {
              brake_status = 1;
              digitalWrite(BRAKE, HIGH);
            } else {
              brake_status = 0;
              digitalWrite(BRAKE, LOW);
            }
          }

          /* Send the pixel value to the device */

          /* TESTING */
          //if ((my_y > 10) && (my_y < 20) && (my_x > 10) && (my_x <
20))
          //  testchar = (unsigned char)10;
          //else
          //  testchar = (unsigned char)0;
          //outputa(testchar);
          //printf("x: %d y: %d y multiplied %d output %x\n", my_x,
my_y, work_y, (unsigned int)testchar);

          outputa(*work_pointer);

          /* bump up the start time for the next pixel */
          baird_pixel_start = time_add(baird_pixel_start,
baird_pixel_duration);
        }
        /* work is done - update times */
        baird_line_start = time_add(baird_line_start,
baird_line_duration);
        baird_pixel_start = baird_line_start;
      }
    }
  }
  pthread_exit(NULL);
}

void *set_baird_delay(void *nothing)
{
  while(1) {
    scanf("%ld", &baird_delay);
    printf("You entered %ld\n", baird_delay);
    sleep(1);
  }
  pthread_exit(NULL);
}

void kittimer(void)
{
  static struct timespec start_time;
  static struct timespec stop_time;
  static long stop_nsec;
  static long start_nsec;
  static long duration;
  /* avg_duration needs to be long long in case totaling for average
*/
  /* results in value over 4 billion (max for long) */
  static long long avg_duration_total;
  static struct timespec duration_timespec;
  static struct timespec working_kit_duration;
  static struct timespec working_kit_delay;
  static struct timespec working_kit_frame_start;
  static struct timespec working_kit_frame_end;
  static long avg_bucket[20];
  static int ct1;
  static int kit_working_frame_count;
  static int last_state;
  static int my_error;
  static int my_direction;

  my_error = clock_gettime(CLOCK_REALTIME, &stop_time);
  if (my_error) {
    perror("clock_gettime");
    exit(1);
  }

  my_direction = time_sub(&duration_timespec, stop_time, start_time);

  /* Corner case if machine first start or glitch */
  if ((my_direction < 0) || (duration_timespec.tv_sec > 0)) {
    duration = (long)1000000;
  } else {
    duration = nsec_time(duration_timespec);
  }

  working_kit_duration.tv_sec = 0;
  working_kit_duration.tv_nsec = (long)duration;

  /* Calculate next frame start; needs to use delay, which is */
  /* percentage of the total duration time */
  working_kit_delay.tv_sec = (time_t)0;
  working_kit_delay.tv_nsec = working_kit_duration.tv_nsec /
(long)100;
  working_kit_delay.tv_nsec = working_kit_delay.tv_nsec * kit_delay;
  working_kit_frame_start = time_add(stop_time,
    working_kit_delay);
  working_kit_frame_end = time_add(working_kit_frame_start,
    working_kit_duration);

  /* now copy in after getting lock */
  kit_frame_count += 1;
  kit_duration = working_kit_duration;
  kit_frame_start = working_kit_frame_start;
  kit_frame_end = working_kit_frame_end;
  kit_delay_time = working_kit_delay;

  /* New Rotation Starting, Move Stop to Start */
  start_time = stop_time;
}

void *scan_kit(void *nothing)
{
  static int my_x, my_y, my_image;
  static unsigned char *bufptr;
  static unsigned char *work_pointer;
  static struct timespec my_kit_frame_duration;
  static struct timespec my_kit_frame_start;
  static struct timespec my_kit_frame_end;
  static struct timespec my_kit_delay_time;
  static struct timespec working_kit_frame_start;
  static struct timespec kit_line_start;
  static struct timespec kit_line_duration;
  static struct timespec kit_pixel_start;
  static struct timespec kit_pixel_duration;
  static struct timespec current_time;
  static struct timespec time_difference;
  static struct timespec time_sum;
  static int time_direction;
  static int vertical_offset;
  static int work_y, work_x;
  static int my_kit_frame_count;
  static int working_kit_frame_count;

  /* Initialize to 0 */
  working_kit_frame_start.tv_sec = 0;
  working_kit_frame_start.tv_nsec = (long)0;

  while(1) {
    /* Continually loop on film */
    bufptr = kit_buf;
    for (my_image = 0; my_image < image_count; my_image += 1) {
      /* on each frame */
      /* set pixel buffer to start of the new frame */
      bufptr = kit_buf + (KIT_ROWS * KIT_COL * my_image);

      /* Now wait for new frame start */
      /* new frame starts when my frame start time is no longer */
      /* equel to system frame start time */
      do {
        /* first wait for timer frame count to increment */
        pthread_mutex_lock(&kit_lock);
        working_kit_frame_count = kit_frame_count;
        pthread_mutex_unlock(&kit_lock);
        if (working_kit_frame_count == my_kit_frame_count) {
          /* timer's kit frame count has not advanced yet */
          sched_yield();
        }
      } while (working_kit_frame_count == my_kit_frame_count);
      /* Now we have new frame count; we still must wait for
         new frame start as we still may be in the delay */
      do {
        pthread_mutex_lock(&kit_lock);
        working_kit_frame_start = kit_frame_start;
        pthread_mutex_unlock(&kit_lock);
```

```
        clock_gettime(CLOCK_REALTIME, &current_time);

        time_direction = time_sub(&time_difference,
          current_time, working_kit_frame_start);
        if (time_direction < 0) {
          /* We are still before the new frame's start */
          sched_yield();
        }
      } while (time_direction < 0);

      /* Now at new frame start */
      /* set new times */
      pthread_mutex_lock(&kit_lock);
      my_kit_frame_count = kit_frame_count;
      my_kit_frame_duration = kit_duration;
      my_kit_frame_start = kit_frame_start;
      my_kit_frame_end = kit_frame_end;
      my_kit_delay_time = kit_delay_time;
      pthread_mutex_unlock(&kit_lock);
      // printf("kit_frame_start count %d sec %ld nsec %ld\n",
my_kit_frame_count, my_kit_frame_start.tv_sec,
my_kit_frame_start.tv_nsec);
      kit_line_start = my_kit_frame_start;
      kit_pixel_start = my_kit_frame_start;
      /* Kit lines are horizontal backwards */
      kit_line_duration = devide_time(my_kit_frame_duration,
KIT_ROWS);
      kit_pixel_duration = devide_time(kit_line_duration, KIT_COL);

      // printf("frame: %ld line: %ld pixel: %ld\n",
      //   my_kit_frame_duration.tv_nsec, kit_line_duration.tv_nsec,
      //   kit_pixel_duration.tv_nsec);

      /* Now wait for the start of this frame in real time*/
      do {
        clock_gettime(CLOCK_REALTIME, &current_time);
        time_direction = time_sub(&time_difference,
          current_time, my_kit_frame_start);
      } while (time_direction < 0);

      // printf("Processing Frame\n");

      /* Now the housekeeping is done for the frame; now output it by
line */
      for (my_y = 0; my_y < KIT_ROWS; my_y += 1) {
        /* Lines are horizontal; starting from top right */
        vertical_offset = my_y;
        /* Lets make sure that we have not run out of time for the
frame */
        pthread_mutex_lock(&kit_lock);
        working_kit_frame_start = kit_frame_start;
        pthread_mutex_unlock(&kit_lock);
        time_direction = time_sub(&time_difference,
my_kit_frame_start,
          working_kit_frame_start);
        if ((time_difference.tv_sec != 0) ||
          (time_difference.tv_nsec != 0)) {
          /* System has new time - lets make sure we are still within
the
             time allocated for this current frame. Grab the system
time;
             we still may be in the delay window */
          clock_gettime(CLOCK_REALTIME, &current_time);
          time_direction = time_sub(&time_difference,
working_kit_frame_start,
            current_time);
          if (time_direction < 0) {
            /* We've shot beyond the time for the new frame; bail out
*/
            // printf("kit turnover at %d\n", my_y);
            break;
          }
        }
        /* now wait for this line's start time */
        do {
          clock_gettime(CLOCK_REALTIME, &current_time);
          time_direction = time_sub(&time_difference,
            current_time, kit_line_start);
        } while (time_direction < 0);
        // printf("line\n");

        /* now output the individual pixels for the line */
        for (my_x = 0; my_x < KIT_COL; my_x += 1) {
          /* work values to point to pixel in memory, which */
          /* is by horizontal line for GIF files */
          work_x = KIT_COL - my_x - 1;
          work_y = vertical_offset * KIT_COL;
          work_pointer = bufptr;
          work_pointer = work_pointer + work_y + work_x;
          /* Wait for the pixel start */
          clock_gettime(CLOCK_REALTIME, &current_time);
          time_direction = time_sub(&time_difference,
            current_time, kit_pixel_start);
          if (time_direction < 0) {
            do {
              clock_gettime(CLOCK_REALTIME, &current_time);
              // printf("pw\n");
              time_direction = time_sub(&time_difference,
                current_time, kit_pixel_start);
            } while (time_direction < 0);
          }
          /* Send the pixel value to the device */

          outputb(*work_pointer);

          /* bump up the start time for the next pixel */
          kit_pixel_start = time_add(kit_pixel_start,
kit_pixel_duration);
        }
        /* work is done - update times */
        kit_line_start = time_add(kit_line_start, kit_line_duration);
        kit_pixel_start = kit_line_start;
      }
    }
  }
  pthread_exit(NULL);
}

void *process_button(void *nothing)
{
  static int my_error;
  static struct timespec this_time;
  static struct timespec difference_time;
  static struct timespec brake_last_pushed;
  static int direction;

  brake_last_pushed.tv_sec = 0;
  brake_last_pushed.tv_nsec = (long)0;

  while(1) {
    /* Has button been pressed */
    if (digitalRead(BUTTON) != 1) {
      /* Button Pressed */
      // printf("Button pressed\n");
      my_error = clock_gettime(CLOCK_REALTIME, &button_time);
      digitalWrite(RELAY, HIGH);
    }

    /* Has brake button been pressed while running */
    if (digitalRead(BRAKE_BUT) != 1) {
      /* Make sure not bounce */
      my_error = clock_gettime(CLOCK_REALTIME, &this_time);
      direction = time_sub(&difference_time, this_time,
brake_last_pushed);
      if ((difference_time.tv_sec > 0) || (difference_time.tv_nsec >
1000000000)) {
        /* Legitimate press, this is not button bounce after one
second */
        if (brake_use == 0) {
          /* Turn brake on */
          brake_use = 1;
          brake_status = 0;
          brake_cycle_count = 0;
        } else {
          /* Turn brake off */
          brake_use = 0;
          brake_status = 0;
          brake_cycle_count = 0;
          digitalWrite(BRAKE, LOW);
        }
        /* Update brake button pushed time to current */
        brake_last_pushed = this_time;
      }
    }
    sched_yield();
    my_error = clock_gettime(CLOCK_REALTIME, &this_time);
    direction = time_sub(&difference_time, this_time, button_time);
    if (difference_time.tv_sec > (1 * 60)) {
      /* Three minutes of viewing - turn off motors */
      // printf("turn off time elapsed\n");
      digitalWrite(RELAY, LOW);
      digitalWrite(BRAKE, LOW);
      brake_use = 0;
      brake_status = 0;
      brake_cycle_count = 0;
    }
    sched_yield();
  }
  pthread_exit(NULL);
}

/* main program; please note that commented code is used for testing
*/
main (int argc, char **argv)
{
  char outchar;
  int img_ct = 0;
  int movie_done = 0;
  size_t pixel_ct = 0;
  size_t row_ct = 0;
  size_t gif_row_size;

  long test1, test2, test3, time_tot, time_avg;
  int counter1, counter2;
  int my_error;

  Display *my_display;
  int my_screen;
  Window my_window;
  GC my_gc;

  filename = argv[1];

  setupgif();
  process_both_movies();

  button_time.tv_sec = 0;
  button_time.tv_nsec = (long)0;

  brake_status = 0;
  brake_use = 0;

  fix_timer_interval.tv_sec = 0;
  fix_timer_interval.tv_nsec = (long)53160000;

  baird_delay = (long)BAIRD_DELAY;

  kit_delay = (long)KIT_DELAY;

  kit_frame_count = 0;

  if (pthread_mutex_init(&baird_lock, NULL) != 0) {
    printf("baird mutex lock failed\n");
    perror("baird mutex lock");
    exit(-1);
  }

  if (pthread_mutex_init(&kit_lock, NULL) != 0) {
    printf("kit mutex lock failed\n");
    perror("kit mutex lock");
    exit(-1);
  }

//  open_display(&my_ds);
//  while (1) {
```

```
//    int xxx, yyy, zzz, pixel_counter;
//    for (zzz = 0; zzz < image_count; zzz += 1) {
//      //display_orig(&my_ds, GifFile->SavedImages + zzz);
//      for (yyy = 0; yyy < BAIRD_ROWS; yyy += 1) {
//        for (xxx = 0; xxx < BAIRD_COL; xxx += 1) {
//          pixel_counter = zzz * BAIRD_ROWS * BAIRD_COL;
//          pixel_counter += yyy * BAIRD_COL;
//          pixel_counter += xxx;
//          do_pixel(&my_ds, xxx, yyy, *(baird_buf + pixel_counter));
//          printf("outputting pixel %d\n",*(baird_buf +
pixel_counter));
//       }
//      }
//    }
//  }

//  do_pixel(&my_ds, 0, 0, 1);
//  do_pixel(&my_ds, 1, 0, 8);
//  do_pixel(&my_ds, 2, 0, 14);
//  sleep(5);
//  exit(0);

  wiringPiSetupGpio();
  pinMode(BITA0, OUTPUT);
  pullUpDnControl(BITA0, PUD_UP);
  pinMode(BITA1, OUTPUT);
  pullUpDnControl(BITA1, PUD_UP);
  pinMode(BITA2, OUTPUT);
  pullUpDnControl(BITA2, PUD_UP);
  pinMode(BITA3, OUTPUT);
  pullUpDnControl(BITA3, PUD_UP);

  pinMode(BITB0, OUTPUT);
  pullUpDnControl(BITB0, PUD_UP);
  pinMode(BITB1, OUTPUT);
  pullUpDnControl(BITB1, PUD_UP);
  pinMode(BITB2, OUTPUT);
  pullUpDnControl(BITB2, PUD_UP);
  pinMode(BITB3, OUTPUT);
  pullUpDnControl(BITB3, PUD_UP);

  pinMode(BAIRD, INPUT);

  pinMode(KIT, INPUT);

  pinMode(BRAKE_BUT, INPUT);
  pullUpDnControl(BRAKE_BUT, PUD_UP);
  pinMode(BUTTON, INPUT);
  pullUpDnControl(BUTTON, PUD_UP);

  pinMode(RELAY, OUTPUT);
  pullUpDnControl(RELAY, PUD_UP);

  pinMode(BRAKE, OUTPUT);
  pullUpDnControl(BRAKE, PUD_UP);


  img_ct = 0;

  digitalWrite(RELAY, LOW);
  digitalWrite(BRAKE, LOW);

  outputa((unsigned char) 0);
  outputb((unsigned char) 0);

  brake_use = 0;
  brake_status = 0;
  brake_cycle_count = 0;

  Error = pthread_create(&button_pthread, NULL, process_button,
NULL);
//  Error = pthread_create(&delay_entry_pthread, NULL,
set_baird_delay, NULL);
  Error = pthread_create(&fix_timer_pthread, NULL, fixtimer, NULL);
//  Error = pthread_create(&baird_timer_pthread, NULL, bairdtimer,
NULL);
  Error = wiringPiISR(BAIRD, INT_EDGE_RISING, bairdtimer);
  Error = pthread_create(&scan_baird_pthread, NULL, scan_baird,
NULL);
//  Error = pthread_create(&kit_timer_pthread, NULL, kittimer, NULL);
  Error = wiringPiISR(KIT, INT_EDGE_RISING, kittimer);
  Error = pthread_create(&scan_kit_pthread, NULL, scan_kit, NULL);

/* TESTING */
  while(1) {
    sleep(1);
  }


//  open_display(&my_ds);
//  while (1) {
//    int xxx, yyy, zzz, pixel_counter;
//    for (zzz = 0; zzz < image_count; zzz += 1) {
```

```
//      //display_orig(&my_ds, GifFile->SavedImages + zzz);
//      for (yyy = 0; yyy < BAIRD_COL; yyy += 1) {
//        for (xxx = 0; xxx < BAIRD_ROWS; xxx += 1) {
//          pixel_counter = zzz * BAIRD_ROWS * BAIRD_COL;
//          pixel_counter += yyy * BAIRD_ROWS;
//          pixel_counter += xxx;
//          do_pixel(&my_ds, xxx, yyy, *(baird_buf + pixel_counter));
//          printf("outputting pixel %d for x %d of %d and y %d of %d
pix count %d\n",
//            *(baird_buf + pixel_counter), xxx, BAIRD_ROWS, yyy,
BAIRD_COL, pixel_counter);
//        }
//      }
//    }
//  }

//  while(1) {
//    for (outchar = 0; outchar < 16; outchar += 1) {
//      outputa(outchar);
//      outputb(outchar);
//      if ((outchar <= 8) || (outchar >=10)) {
//        outputb(15);
//        outputa(15);
//      } else {
//        outputb(0);
//        outputa(0);
//      }
//      for (img_ct = 0; img_ct < 30000; img_ct += 1) {
//        pixel_ct = 0;
//      }
//    }
//  }
}
```