Serial Communication/Protocol

All of the sensors used to monitor the functions of the coffee pot, and the camera for taking mug shots are going to be controlled with a dedicated PIC microcontroller. The PIC will monitor the sensors and perform preliminary processing on the sensor data. The PIC will need to send this data to the SNAP microcontroller for final processing and permanent storage of the data on the web page. In order to send the sensor data between the PIC and SNAP, we chose to use a bi-directional serial communication line using a standard interface. With this type of communication, we will be able to quickly send data between the two microcontrollers, and use only two I/O pins.

The serial communication requires a standard protocol to send and retrieve data, so we have created a simple communication protocol between the two microcontrollers. In the initial protocol, the PIC is used exclusively for sending data and the SNAP is used exclusively for receiving data. This protocol is shown in Figure 1.

![Figure 1 - State Diagram of Sensor Communication Protocol](image)
As an example (from the state diagram above), when the PIC wants to transmit an image to the SNAP for final processing and saving, it will:

1. Send a single byte of value 3. This will tell the SNAP microcontroller that an image is about to be sent.
2. Send a byte for the width of the image, followed by a byte for the height of the image.
3. Send a byte indicating the image format (grayscale, RGB format, or YUV format).
4. Send each pixel byte in order (for grayscale 1 byte per pixel; for 24-bit RGB, 3 bytes per pixel).

Using the flow chart above, it can also easily be seen how to send data taken from each sensor.

When initially implementing the protocol for the image data, we had a few problems with getting buffer overflows on the serial input buffer of the SNAP. We found that when trying to send the image data at a baud rate of 28800 or greater, the SNAP would get a buffer overflow and throw an exception. However, if we used a slower baud rate, the data image would transmit correctly, but would take excessive amounts of time to transmit (for example, 2400 baud took 67 seconds to transmit a 128 x 128 grayscale image). It turns out that the problem was in the SNAP allocating the array needed to hold the image. This allocation would block the serial input for a long enough time that the buffer overflowed. In order to correct this, we modified the protocol so that the PIC would send the image header portion (everything except the pixels), and wait for the SNAP to respond with an acknowledgement message (just a byte containing the width of the image expected), and then the PIC would send the actual image data. Thus, both microcontrollers need to send and receive data serially.

So far we have not been able to come up with a good means of recovering from an error in the serial protocol. Errors could range from something like sending a response out of the range of expected values. For instance, if you are in the initial/idle state, and the PIC sends something bigger than a 3, then the SNAP board will not know how to handle this event. We need to come up with a way of handling these unexpected events in a way that will minimize the loss of data.

**Camera**

A current 128 x 128 pixel grayscale camera image using the GameBoy camera is shown in Figure 2, which is a picture of a desk with a keyboard and paper on it, and a felt chair and whiteboard in the background. This image was captured from the sensor microcontroller and transmitted serially to the SNAP through RS-232, at a rate of 115200 baud.

![Figure 2 - Current Camera Picture](image-url)
Porting to PIC

The image capture algorithm was developed on a Cygnal development board; the Cygnal facilitated faster protocol development than the PIC processor, which we intend to use in the final product for its smaller footprint and lower cost. Work is underway to port this code to the PIC. We’ve been successful in transmitting serial data to the SNAP from the PIC, and image capture is buggy but working. Our biggest remaining task is to get serial read working; the PIC is currently returning garbage when we attempt to read from the serial port. If reading proves to be problematic, we have a contingency plan to use available pins on the SNAP to send interrupts to the PIC. This precludes some functionality, such as web-based access to camera registers and raw sensor data, but will still enable us to deliver our core set of features.

Sensor Mounting

In order to mount the phototransistors/LED’s on the coffee pots site tube plastic pipe hangers will be utilized. Figure 3 shows the type of pipe hangers that we plan on using for mounting the phototransistors/LED’s to the site glass.

![Figure 3 - LED Mounting](image)

During the early part of this week we will be purchasing the mounting brackets from a hardware store. We will then be able to test mount the sensors to see how our design is going to work.

It has also been determined that we need to know when the coffee is being brewed in order to determine the level of the coffee. It appears that people begin taking coffee from the machine as soon as they start brewing a fresh pot. So the level of the coffee in the site tube fluctuates quite a bit when they start brewing a fresh pot of coffee. To alleviate the sensors on the site tube from reading these fluctuations, we need to know when the coffee pot is in a brewing state. As of right now, the best way to determine when the coffee pot is brewing is to determine when the pump in the machine is running. In order to do this we would have to tap into the power for the pump and convert this voltage down to a usable signal. This could be accomplished with a simple relay. However, it would involve removing the cover of the coffee pot to install. So we will need to get this approved.

One last item that needs to be considered for the mounting is how to mount the microcontrollers. Currently we plan to get a project box and cut out holes for the connections such as power and Ethernet. The PIC and SNAP microcontrollers will both be mounted in the same box to help minimize the layout of our project.
SNAP Code Optimizations

Three functions have had significant optimizations in the past two weeks: image compression, image storage, and page generation.

Image Compression

First, a brief introduction to GIF compression: GIF images are limited to 256 colors. Each color in the image is converted to an indexed color (since our images are currently grayscale, this is trivial) and then LZW compression is performed on the resulting indexed colors. LZW compression is essentially Huffman encoding of the image data, in which strings of characters are mapped to shorter bit patterns. To facilitate this, a hash table is kept with the strings encountered thus far and their corresponding bit pattern. This hash is typically implemented as a simple array with a stepping hash function. In the GIF implementation we are using, this hash function is implemented as a static class method which is called once for each pixel being compressed; the hash function is fairly simple to compute, but benchmarks showed that a large amount of time was being spent in the function, so optimization efforts were focused here, and the results can be seen in Table 1.

The first attempt was increase the size of the hash table in an effort to reduce hash collisions and speed up searches. Two parameters control the size and operation of the hash table: HASHSIZE and HASHSTEP. HASHSIZE was increased, with no effect, followed by an increase in HASHSTEP, again with no effect. Since this appeared ineffective, the changes were rolled back out of the code.

It seemed that a lot of overhead would be involved in making repeated calls to a small static method, so we next made the hashing routine inline code (since it is only called in two places, this was a straight forward modification). This was much more effective, yielding an almost 30% increase in compression speed, as shown in Table 1.

<table>
<thead>
<tr>
<th>Optimization</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>LZW Compression, baseline</td>
<td>3.40s</td>
</tr>
<tr>
<td>HASHSIZE = 30011</td>
<td>3.40s</td>
</tr>
<tr>
<td>HASHSIZE = 30011, HASHSTEP = 7717</td>
<td>3.40s</td>
</tr>
<tr>
<td>Inlined Hash() method</td>
<td>2.34s</td>
</tr>
</tbody>
</table>

Table 1 - LZW Compression Optimizations

Image Storage

In order to get the GIF encoder working, we had to implement a subset of the Java AWT’s Image subsystem. This was a naïve implementation which packed the RGB values into an array of ints; the GIF encoder, however, then turned around and unpacked these values into separate red, green, and blue channels. We added a non-standard method to the Image class which permits direct access to these channels, and which now only packs them when packed versions are requested. The results are shown in Table 2.

<table>
<thead>
<tr>
<th>Function</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>RGB packing</td>
<td>0.45s</td>
</tr>
<tr>
<td>RGB unpacking</td>
<td>0.54s</td>
</tr>
</tbody>
</table>

Table 2 - Image Packing Running Times
With the new implementation, both of these operations are unnecessary, yielding a 1 second improvement to each image creation process.

**Page Generation**

As you will no doubt recall, our previous efforts at improving page generation time were focused on improving the search and replace of placeholder strings. Last week, we decided to scrap this approach in favor of a more convoluted, but ultimately much more efficient, technique.

This new technique involves converting each page into a block of constant strings concatenated with the appropriate variables. For example, the following excerpt from our template file (these examples are not taken directly from our source code; they merely demonstrate the process):

```html
…<td><div class="bar">|CupsPouredMonday|</div></td>…
```

might get translated into this Java code:

```java
…"<td><div class="bar">" + cupsPouredForDay[MONDAY] + "</div></td>"…
```

While this would be a tremendous hassle were it done by hand, a perl script was written that performs this conversion automatically.

Running times for our initial optimizations are included in Table 3 for reference; note that we have achieved approximately 90% increase over the previous optimal time. Benchmarks were performed on the generation of the index page only; CSS generation has not yet been integrated. However, since the CSS file is much smaller, we expect comparable running times.

<table>
<thead>
<tr>
<th>Optimization</th>
<th>Running Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>No optimizations</td>
<td>9.14s</td>
</tr>
<tr>
<td>StringBuffers</td>
<td>8.99s</td>
</tr>
<tr>
<td>StringBuffers/Offset hashing</td>
<td>9.76s (first pass)</td>
</tr>
<tr>
<td></td>
<td>1.36s (subsequent passes)</td>
</tr>
<tr>
<td>Inline generation (index only)</td>
<td>0.10s</td>
</tr>
</tbody>
</table>

*Table 3 - Page Generation Optimizations*

**Other Software Accomplishments**

The daylight saving time change came at a fortunate time; after April 4, all of our times were off by one hour. This necessitated several modifications; currently, the software is hard coded to convert times from GMT to U.S. Central Standard Time. If this code is distributed to others, a modest amount of effort will be needed to generalize the time conversion functions. The good news is that our date/time routines are now significantly more robust.

Automated test suites have been written for the image-related classes; these appear to be working well, and were invaluable during testing of the optimizations discussed earlier.
Remaining Tasks

The following are the major tasks which are not yet complete:

- **Hardware**
  - Wiring a relay or similar device to the hot-water pump inside the coffee maker to detect the starting and stopping of brewing.
  - Mounting the phototransistors to the coffee level tube.
  - Designing the case to house our entire system near the coffee maker. This must be unobtrusive, functional, durable, and portable.

- **Software**
  - Mugshot gallery – the new inline page generation code is not as well suited to generating the kind of HTML needed by the gallery, and these issues have yet to be worked out. Implementation is scheduled for this week. Note that mugshots are being saved properly, we just aren’t generating a web page for them yet.
  - We are planning on splitting off the image compression code into its own thread so that the SNAP is not blocked for the several seconds required to write the image. This code is fairly self-contained, and we expect to need very little thread synchronization, so work should proceed quickly.
  - We need to split the generated content out into sub-pages, as well as add the e-mail notification feature. Both of these tasks are underway, but were put on the back burner while the serial transfer issues were worked out. We expect to attend to them shortly; if necessary, work on these can be performed after deployment of the system.
  - Load testing on the web server.