SE3910 Lab 8: Building a Video Conferencing Application

Due: May 16, 2014 (Schilling's Section)

Due: Week 10 Wednesday, 11pm (Yoder's Section)

1. Introduction

The last lab is here. But, by the same token, the most important lab is here, for in this lab, you will put together all of the tasks that you have learned into a final project. The project will involve sending images across the network to a second GUI, and showing them. You will use sockets. You will use real time analysis. And you will have to do some design, as well as verification that your system is working properly.

Because of the increased scope of this lab, and the timing with the end of the quarter, this project may be completed in larger teams of up to 4 people. However, every team member is responsible for understanding the entire project and being able to explain what is occurring in the entire system. And there may be questions related to any portion of the implemented system.

This is also a multiweek lab. The final report will be due Friday of week #10 for Dr. Schilling's Section and Wednesday of Week #10 for Dr. Yoder's Section.

2. Lab Objectives

- Design a system to meet a given set of real time constraints.
- Use QTCreator to create an application which can run on both the desktop and on a remote embedded target.

3. Initial steps

In order for this lab to work properly, you will need to install one package on your Beaglebone. To do this, issue the following command:

```
sudo apt-get install time
```

This will install a necessary timing utility on your beaglebone.

4. Part 1: Measuring how long it actually takes to capture an image

In the past, we've spent a lot of time capturing a single image. But in doing that, there is a ton of overhead involved in setting up the stream and shutting down the stream. We need to try and get an idea of how long it takes to do this setup and teardown as well as how long it takes to capture each individual image.

```
gst-launch v4l2src device=/dev/video0 num-buffers=10 ! ! video/x-raw-
yuv,framerate=10/1,width=320,height=240 ! ffmpegcolorspace ! jpegenc !
multifilesink location =”frame%05d.jpg”
```

The command above will work on your Beaglebone to take a set of 10 pictures. On the course website, there is a pair of scripts, testAllResolutions.sh and testCamera.sh. The scripts will measure how long it will take to capture images from the camera at various resolutions between 320 x 240 and 1280 x 720. Download these scripts to the Beaglebone and execute them to measure the performance of your system.

Run the command **sudo apt-get install time** to install the fancy form of the time utility used by these scripts.

Create a series of plots, one for each resolution, which show on the horizontal (“x”) axis the number of images captured and the vertical (“y”) axis the time.

This data should roughly follow the linear relation

$$TotalTime = t_{sSetup} + n \times t_{capture}$$

where n is the number of frames captured, $t_{capture}$ is the time to capture a single frame, and t_{setup} is the time to create the GStreamer pipeline. Fit a trendline to the data using Excel. Be sure to check the “include equation on graph” option which will include an equation very similar to the one above. From these plots and equations, calculate t_{setup} and $t_{capture}$ for each of the four major resolutions. (The intercept of the each plot represents the setup time for that resolution.)

Once you have this data, determine the processor utilization while this work is occurring. The utilization is the sum of the user time plus the system time divided by the elapsed time. For each one of these items, plot the utilization versus the number of frames taken. Note that this data has all been taken while capturing at a nominal 5 frames per second. ***You can include all the plots on the same axis, as long as the various point-sets and lines can be distinguished clearly.***

Given the data and the performance, is it possible for your system to reliably receive data from the camera at 5 frames per second if the processor utilization cannot be more than 50 percent for this task?

5. Part 2: Modifying your QT Application to capture images and show them when they occur.

The first thing we need to do is to modify our QT application from two weeks ago (Lab6). We are going to modify the application to have two new menu items, start, and stop. Each item will also have new signal handler which will be invoked when the menu item is invoked.

You may also want to add a new slot to open a file by filename (bypassing the default image selection dialog box) as well as a slot to open and delete a file by filename. You’ll want to look at the information at http://qt-project.org/doc/qt-4.8/signalsandslots.html to explain these concepts as well as to show an example.

The start operation is responsible for performing the following. First and foremost, it will start a thread running which will start capturing from the video source and storing the files as jpeg files in sequential order. The frame rate to be captured should be 5 frames per second, and the resolution should be that which was determined in Part 1. It will also start a second thread. This thread will look for files which are new and have been written by the first thread to the disk. It will open the most current thread into the display, delete the file, and then will sleep for a given period of time representing the delay until the next image is expected.[1]

In developing your program, you should target using the virtual machine as an initial test platform. To do this, you will need to use a test source instead of an actual camera source. Here is an example gstreamer command that you might run as a system call (command-line call) from your program

gst-launch videotestsrc num-buffers=50 ! video/x-raw-yuv, framerate=10/1, width=320, height=240 ! ffmpegcolorspace ! queue ! jpegenc ! multifilesink location="frame%05d.jpg"

Your GUI must remain responsive while the video is playing. You may want the video producer to execute in its own thread. There are several ways you can accomplish this. The first is to use good old fashion posix threads. You also can use the QT QThread class, which is extremely similar to how Java handles threads. (See http://qt-project.org/doc/qt-4.8/qthread.html and http://qt-project.org/doc/qt-4.8/thread-basics.html#which-qt-thread-technology-should-you-use)

In doing this testing, you should use a conditional compile that will allow the program to be rebuilt for either the development environment or the target environment by changing a single definition. If using development, the image will be the test image, while if on the target, the image will be the actual image captured from the camera.

You also will need to have a dialog box showing how many frames have been transmitted as well as how many bytes have been sent to the server. On the server, there should be a dialog box showing how many frames have been received and how many bytes have been received.

6. Part 3: Transferring images to the second beaglebone / development machine

Once you have started making the images occur reliably occur on the local machine, we want to send them to a second machine. This will involve using QTSockets to send the image across the network to a second machine.

You'll need to make a few changes in your code. First and foremost, when you start the process as a client, you'll need to be able to enter in the ip address and port for the server that is to receive the image. You should allow the user to pass this info along as a command line parameter.

[1] Astute students will note that writing to files is not the most effective way to handle this type of a system, and you are correct. However, there are issues with the GStreamer interface to QT that prevent us from being able to reliably stream directly into memory on the image on your beaglebone.

To do this, you'll want to look at bit at QT Sockets. The page http://www.hanckmann.net/?q=send_an_image_over_a_network_using_qt has some example code that shows how to do this.

You'll end up building a second GUI that is the server (or having a startup configuration flag that is passed on the command line to indicate whether the program is to run as a client or a server).

When this is done, you should be able to see images passed from the client to the server in real time.

7. Part 4: Verification of your system

The final step of this lab is to verify that the operational profiles you defined in the first phase are actually being met. You will need to measure with an oscilloscope how often an image is sent from the client to the server, as well as how long it takes for the display to refresh when a new image is acquired. You'll also need to ensure that the real time performance of the system is met. You'll need to figure out what the processor utilization is for the task which captures the image from the camera, the task which updates the UI, the task which sends the image across the network, the task which received the image from the network, and the task which updates the image on the screen. Using the time utility, you'll also want to measure the overall utilization of the CPU. You will need to verify that the frame rate is being met reliably by the system, or if it is not, indicate how often there is a deviation from the desired performance criteria.

8. Deliverables / Submission

Each team will be responsible for submitting one report with the following contents:

1. Introduction -> What are you trying to accomplish with this lab? This section shall be written IN YOUR OWN WORDS. DO NOT copy directly from the assignment.
2. Initial Analysis
   a. Include a table showing your raw data with the resolution of the images and the times that it took to capture images, as well as the utilization.
   b. Include a table summarizing the setup time and image capture time for each resolution captured by the script.
   c. Include one or more plots showing the amount of time taken to capture the images for all resolutions
3. Include a plot of the utilization versus the number of images captured.Rate Monatonic Analysis
   a. Based on the data you measure in step 2, perform a rate monotonic analysis on the tasks that will be present in your system. What are the task priorities, and how often do they execute?
   b. Do you have the ability to add an additional task, audio task, which will send an audio channel (supposing there was one available on the bone) from the client to the server using a second parallel socket? If you need to read an audio device 100 times per second, where would this fit into the priority scheme?
4. Project Allocation
   a. Include a plan, shown to professor(s) by the end of the Week 8 Lab, explaining how the work of your project is to be divided amongst the team members. Each team member must be given specific tasks that they are responsible for completing on the lab.

   b. Effort log. For each task item, record how long it took to complete the task (effort) by the team member.
5. Design
   a. How did you partition your code?
   b. Include brief, high level UML diagrams showing any additional classes you added to the system.
6. Testing Plan
   a. Document which measurements you are going to take to verify that the system is going to operate properly.
   b. Document which debug operations your code will have to allow development on a virtual machine.
7. Test results
   a. With the code executing on the beaglebone, perform the test plan to determine if the project is meeting its specifications.
   b. If there are deviations from the performance specification, note which area of deviation have occurred.
   c. Include an answer to the question at the end of Part 1. Provide support for your answer.
8. Example photos / screen captures
   a. In your report, include a screen shot of the program loaded on your virtual machine showing your name on the title bar.
9. Include a screen shot showing the image viewer running on the embedded host opening a picture which was captured with last weeks program.
10. Things gone right / Things gone wrong -> This section shall discuss the things which went correctly with this experiment as well as the things which posed problems during this lab.
11. Conclusions ->
    a. What have you learned with this experience?
    b. What improvements can be made in this experience in the future?
    c. Key Take away – Each team member shall submit a paragraph describing their key takeaways from the lab.
12. Appendix -> Source code (Dr. Schilling's section only)
    a. For each program, include the source code you used. Code should be well commented and documented.

Additionally, each team should submit an mp4 video file captured with a cell phone or other video device showing the embedded system operating properly, which images being transmitted from one device to the other device. The videos should be no longer than 2 minutes in length, and should include narration explaining what is occurring on the system.

Dr. Schilling's Section

You also should submit a tar file which can build of your project. There may be multiple pieces if your client and server use different code bases.

Dr. Yoder's Section

This material should be submitted to the website as a .pdf, .mp4, and .zip. The pdf should contain the report and NOT the source. The .mp4 should be the video. The .zip file should contain the source code, **and be structured as below**.

The zip file should be structured as follows: (Note that the source is ***not*** deeply nested.)

- lab8.zip
    - lab8
        - README.txt
        - Makefile (if "pure" make used)
        - qmake file (if "qmake" used)
        - Eclipse folder ***without*** Debug folder (if Eclipse used)
            - Source files (If Eclipse used)
            - Source files (If Eclipse used)
        - Second Eclipse folder ***without*** Debug folder (if Eclipse used, second program used)
            - Source files (If Eclipse used)
            - Source files (If Eclipse used)
        - Source files (if Eclipse not used)
        - Source files (if Eclipse not used)

In all cases, there should be a README.txt file in top level of the zip file which can be describes how to build the project.
If you have any questions, consult your instructor.

Version 1.0