Lab 1: Introduction to Gumstix and Code Optimization

18–342 Fundamentals of Embedded Systems

Released: September 25, 2012, 11:59 pm EDT
Due: October 9, 2012, 11:59 pm EDT

Contents

1 Introduction to Part 1 of the Lab

1.1 Overview

This part of the lab is meant to introduce you to the hardware development platform and to show you some good coding practices. You and your partners will get acquainted with the development platform that we will be using throughout the course. You will also learn to use a revision control system. You are strongly encouraged to continue using some form of revision control throughout the rest of the course. Part 1 of this lab is meant to be neither difficult nor time consuming. However this part of the lab is important, as the skills it teaches will be assumed knowledge for future labs. Of course, we also hope that it will be fun.

1.2 Tasks

In this part of Lab 1, you will perform a number of small tasks to get you warmed up and ready for further, more involved development in the class. The embedded system in this class will be the gumstix basix+robostix (verdex-pro+robostix for some of the students) system [15]. You will familiarize yourself with this system. To this end, you will:

1. Create and use a source repository for your team 2.5.
2. Check out files from an SVN repository *(15 points)* 2.6.
3. Establish a serial connection from a provided gumstix embedded system to your computer 4.
4. Send it multiple files of different formats 4.4.
5. Boot a small GNU/Linux system on the gumstix 5.
6. Personalize the GNU/Linux system that you will use for the rest of the semester 5.2.
7. Program a simple Hello World program for the gumstix's ARM processor in C *(20 points)* 6.1.
8. Program a simple program for the gumstix's ARM processor in ARM assembly *(20 points)* 6.2.
9. Disassemble and analyse binaries for the ARM *(20 points)* 6.3.
10. Write a Makefile to automate the build process *(20 points)* 7.
11. Have fun!
12. Some points will be awarded for proper compliance to instructions, proper naming of files and general quality of handout. These are discretionary points for TAs *(5 points)*.

1.3 Traditional Embedded Systems

Traditional embedded computers are low power, low memory devices. Typically chosen to perform a specific task, embedded computers execute highly specialized software designed to function within the constraints of the devices. In part due to these constraints, embedded systems software is almost always developed offline, often on a workstation that features a full software development kit (SDK) including an editor, compiler, hardware simulator, and debugger.

1.4 Modern Embedded Systems

In the past few years, embedded computers have experienced an exponential growth in speed and memory capacity. While many embedded systems still rely on traditional 8-bit microcontroller designs, embedded systems designed for communication or entertainment (cell phones, PDAs, handheld game consoles, etc.) feature 32-bit processors that can outperform desktop machines of ten years ago. With flash memory offering persistent storage in the gigabyte range, it is now possible to natively develop embedded applications on the embedded systems themselves.

1.5 Embedded Systems for this Course

In this class, we will consider both the traditional and the on-system approaches of developing for embedded systems. Some exercises will involve developing code offline while others will involve developing, compiling and testing code on the gumstix hardware itself. In this lab, exercises will be carried out on the gumstix boards. To successfully complete this lab, each group requires access to a computer with a USB port and software to perform terminal communication such as minicom or HyperTerminal. All other hardware resources are provided to you by us in the form of a hardware kit.

2 Revision Control

2.1 Introduction

A revision control or version control system is a toolset that assists programming teams in maintaining the history of and tracking changes made to a project that they are working on. It is an essential tool for even two person teams when projects require multiple iterations of design and testing. In this section, we shall describe the basic terminology related to a revision control system, how to set up a code repository, how to share work with your partners, how to track your partner's progress and how to work together, in parallel.

2.2 Basic Operations

All revision control systems have a concept of the 'set of things it is in charge of'. This set is usually a set of files or objects that the user has entrusted to the revision control system. This set of files is called a *repository* or a *depot*. Revision control systems track all changes made in a repository by different users. At regular intervals, the user commands the repository to take a snapshot of all the changes that have been made. The tool now considers the repository to be at a new *version* or *revision*. Changes such as what files have been added, what files have been removed, which lines have been changed and who changed them are all recorded. This operation is known as a *commit*. Suppose that a hardworking 18–342 group made a number of changes to their repository at 5 am. The next day, they notice that they had mistakenly overwritten an important file. Thankful that they had used a revision control system, they ask the tool to undo all of the changes they had made. This is known as a *revert*. Now consider two 18–342 students who live off campus and work independently. Instead of manually sending each other code, they each try to commit code to the repository that they have set up. They notice that their changes overlap and that they need to cleverly adjust their edits to not overlap. This operation of bringing together independent work on the same file is known as a *merge*. Any overlap in the code that is being merged causes a *merge conflict* that must be manually *resolved* by the merging user.

2.3 Client-Server Model

A client-server model of revision control is the traditional revision control model. The model consists of a primary server that hosts the repository or depot. Users never directly edit the repository. They *check-out* a copy of the code out of the main repository. They then edit their *local* or *working* copy to their hearts content. When they are ready to show the world the work they have done, they commit their code. But before they commit their code, they need to make sure that some one else hasn't changed the repository. Hence, they need to *update* their working copy and resolve any merge conflicts. They then commit their code to the main repository. This model uses minimal space on client systems and has a canonical version history for the entire repository. This model is used by traditional version control systems such as CVS [6], SVN [3] and Perforce.

2.4 Distributed Model

The upcoming models of revision control are all based on distributed, non-central models. In these models, there is no one main repository. Each user's codebase is a bona fide repository in and of itself. Users can *clone* each others repositories to get each other's working copies. Fundamental to this concept is the

concept of *branching* and independent histories. Since each user has their own independent repository, they each maintain their own version of the file histories. The independent arcs of commits are known as *branches*. A branch in the repository signifies alternate histories of what happened to a repository. Branches can be merged if they share a common version in the past known as an *ancestor*. Since multiple versions of a repository and its history are in flight at the same time, developers can easily choose which version they want to work on and who they want to merge with. The tip of a branch that a developer is working on is known as the *head* of that branch. Since branching and merging are such common operations in a distributed model, special tools are used to deal with these. The most common is the three way merge used by Mercurial [17] and Git [23]. A more advanced and formal system called patch calculus is used in Darcs [22].

2.5 Introduction to Git

In this section, we shall introduce you to a popular revision control system called **git**. Git is a widely used, distributed model, multi-platform revision control system. It is already installed on the ECE cluster machines and on all Andrew Unix machines. Clients for it are available online [23] including versions for Windows. We shall now go through a list of basic tasks and commands that you must get familiar with. The tasks described can be performed on the ECE or Andrew Unix systems although they can be easily adapted to other environments as well.

- To create a new repository, use the `git init` command.

  ```
  % git init --bare /path/to/repository/here
  ```

  This directory will now act as the central storage area for all the code in your project. Do not ever manually add, remove or modify files in this directory unless you want to seriously damage your repository. Also note that the files that you work on in the project will not be stored in plain text in the main repo. Do not go looking for them.

- To actually start creating and editing code, you will first need to get a *working copy* of the (currently empty) repository. This is the aforementioned *clone* operation.

  ```
  % git clone /canonical/path/to/repository/here localcopy
  ```

  The above command will create a working copy of the repository in the `localcopy` directory. If you are cloning a repository from a different machine, then use:

  ```
  % git clone ssh://[user@]host.xz[:port]/path/to/repository localcopy
  ```

- You are now free to edit your code, add files and, in general, carry out your work in the `localcopy` directory.

- After you have done some work, it is now time to tell the repository to track the changes you have made. Add the files for commit by issuing the following command.

  ```
  % git add file1 file2 file3
  ```

- This will add your files to a staging area for the commit. If invoked interactively as shown in the following command, this area allows to add entire files, parts of files, just certain chunks of code and also provides various other alternatives.

```
% git add -i
```

- To tell git to take a snapshot of your current progress as a *commit*, issue the following command.

  ```
  % git commit
  ```

  This command will bring up a text editor to allow you to properly and informatively label the changes you are going to commit. The editor that it brings up depends on your `EDITOR` environment variable. This variable is usually set in your `.bash_profile`, `.cshrc` or equivalent file depending on your shell. To find out what shell you are currently using, run:

  ```
  % echo $SHELL
  ```

  You can also temporarily change the `EDITOR` variable by using "`export EDITOR=path/to/editor`" for bash or "`setenv EDITOR path/to/editor`" for csh.

- You can now review your commits and changes using the `git log` command. You can also get elementary help by using the `git help` command.
- This saves the changes as a state / commit in your local repository. You can use the SHA-1 hash of each commit to come back to that state of your repository.
- To make your changes available in the central repository, issue the following command -

  ```
  % git push origin master
  ```

- The steps described above are extremely elementary. Git allows you to perform time-travel through your repository state using `git checkout`. It also allows you to have different parallel universes of your code using branches. **You should read at least the first 2 chapters of the online Git book [2] for a more complete description on how to import your teammate's code, how to merge changes, how to revert any mistakes that you may have made, travel through time in your code and have parallel universes and also to finish the exercise at the end of this section.** Github provides a very good interactive tutorial [14] where you actually create a repository and perform various operations on it. Gitflow [7] is an interesting article on how to use per feature git branches and how you can use git to manage your project.

  Warning: Git is a very powerful system which includes the ability for other people to do many things that you, personally, should not do. Some commands can leave your repository in a mess if not used with extreme caution. These commands include:

  git reset

  git revert

  git rebase

  It is also not a good idea to use –hard or –force parameters without knowing what you are trying to do. As a general rule, do not try to change commit history after the commit has been pushed to the central repository.

  The files in the repository in the exercise below have HOWTOs to create a repository on andrew machines and adjust the directory permissions so that only you and your teammates can access and edit the files in the directory.

2.6 Revision Control System Exercise (15 points)

We have created an git repository at `/afs/andrew.cmu.edu/usr17/ckamat/public/18342_lab0_repository/` which you have read-access to. Clone this repository and then answer the following questions. Record your answers in a plain text file called `repository.txt`.

1. How many branches are there in the repository ?
2. Discounting hidden files, what are the files that are in the master branch ? What are their names? What are their contents?
3. Which one of your TAs/instructor committed those files?
4. When did (s)he commit them (date and time) and what was the commit message?

Warning: Github, Bitbucket and various other sites provide free hosting for git repositories. However, the default settings on these services will make your repositories (and your code) public. This is against the course policy. If you choose to use these services for hosting your code, it is your responsibility to make sure that the repositories are private and visible only to your team (you could apply for student accounts.)

3 Lab Hardware

3.1 Hardware Kit Contents

Each lab group has been provided with a hardware kit with the following components:

- gumstix basix (or verdex-pro) 400xm-bt motherboard
- gumstix robostix expansion board
- gumstix 5.0 volt power adapter
- Acroname USB serial interface connector
- Serial extension cable
- USB A–B cable
- 1 GB MMC (or microSD for verdex-pro)
- PXA–AVR UART jumper

The heart of the 18–342 hardware kit consists of the basix motherboard and the robostix expansion board (see Figure 1). The basix motherboard is the smaller of the two boards and contains an Intel PXA255 processor (now owned by Marvel), 16 MB of onboard flash memory (StrataFlash), an MMC slot, and a Bluetooth module (The verdex-pro motherboard contains an Intel PXA270 processor, 16 MB of onboard flash memory, a miroSD card slot, and a Bluetooth module). The robostix board is the larger of the two and contains an Atmel ATmega128 microcontroller with headers for nearly every I/O pin, as well as additional headers for some of the gumstix (PXA) I/O pins.

The two boards may be connected by the white, 60-pin hirose connector. When connected, the basix/verdex motherboard sits in the center of channel of the robostix between the upper three rows of I/O headers and the power jack. Since the basix/verdex motherboard itself lacks a power connector, it must always be used in conjunction with the robostix board to receive power.

The third PCB is the USB serial interface connector. It may connect to any of the gumstix (PXA) or robostix (AVR) UARTS, but is most often used to connect the Linux console on the gumstix to a host PC.

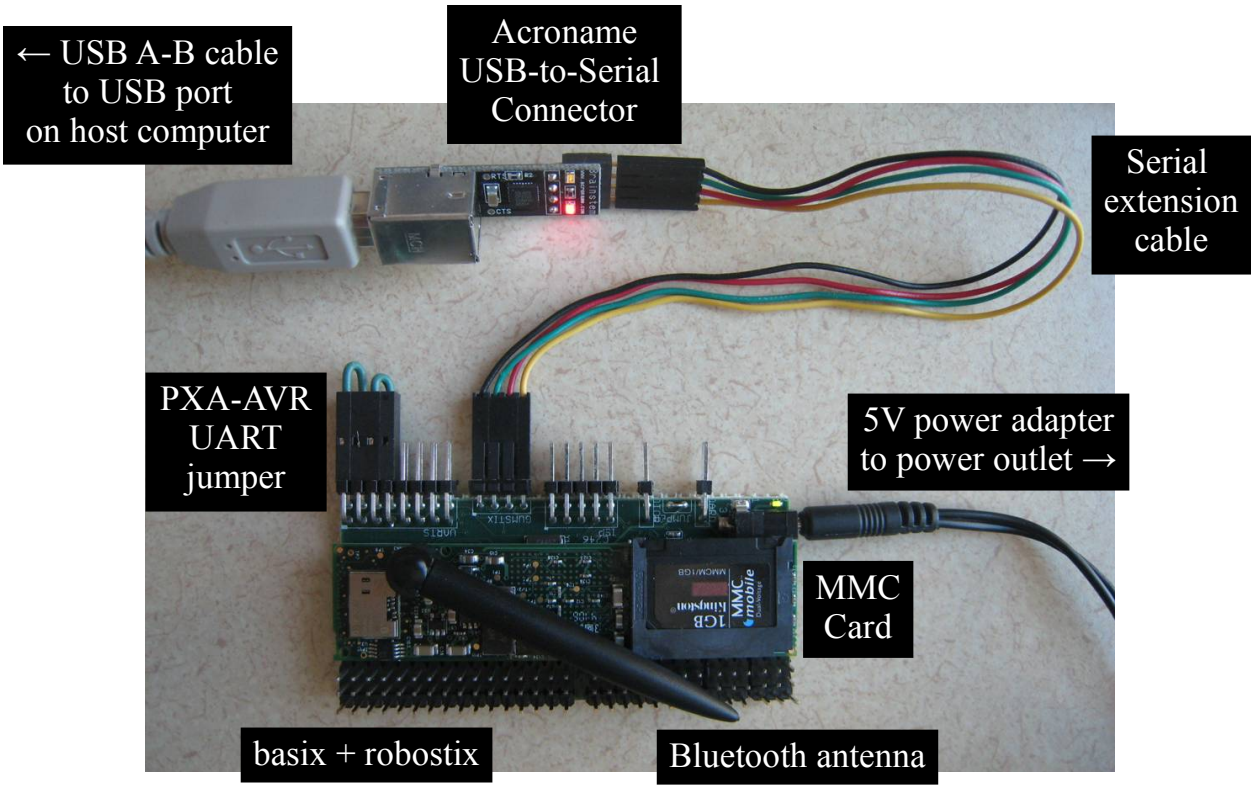

Figure 1: Hardware kit components connected together.

3.2 Hardware Setup

To setup the 18–342 hardware:

1. Remove the card extender from the MMC/microSD and insert the MMC/microSD into the card slot on the motherboard.
2. Gently snap the basix/verdex motherboard into the robostix so that the motherboard lies entirely on top of the robostix (as opposed to hanging off the other side).
3. Attach the PXA–AVR UART jumper on the bottom right set of "UARTS" headers on the robostix board. The jumper should be connected so that the "PXA ST" RX & TX pins connect to the "Atmel 0" TX & RX pins.
4. Connect the male side of the four wire (black/green/red/yellow) extension cable to the "Brainstem" port of the serial converter. The black wire should connect on the left side when the converter is oriented such that the word "Brainstem" reads properly (i.e., Brainstem port on top, USB port on bottom).
5. Connect the female side of the four wire extension cable to the "GUMSTIX" pins on the robostix board. The black wire should connect on the right side when the robostix board is oriented such that the word "GUMSTIX" reads properly (the pins are on the bottom).
6. Connect the USB A-B cable first to the serial converter, then to a USB port on the host computer. If the host computer is running Windows XP, Windows Vista or Mac OS, you may need to install the latest FTDI Virtual COM Port Driver [12] before or while completing this step. The FTDI driver should already be available on most Linux machines.

Note: Do *not* connect the gumstix power adapter until after setting up serial communications on the host machine as described in the next section.

4 Communicating with the Gumstix

4.1 RS–232 Serial Communications

The principal means of communicating with the gumstix boards is through an RS–232 serial connection. RS–232 is a nearly forty-year old standard for serial data signals between computers and peripheral equipment such as terminals, modems, printers, and mice. Until a few years ago RS–232 serial ports were found on nearly every desktop & workstation computers. Since then, the rising popularity of RS–232's successor protocol, the Universal Serial Bus (USB), has seen the elimination of RS–232 ports (along with other legacy I/O) ports from consumer machines. However, due to the simplicity & robustness of the protocol, RS–232 is still heavily featured in both embedded and enterprise systems.

In the context of the gumstix, RS–232 is used to communicate with the gumstix U-Boot bootloader and Linux console, much as a keyboard and monitor is used to communicate with a Linux desktop machine. A Universal Asynchronous Receiver/Transmitter (UART) serves as a "serial controller," and is used by software programs to convert parallel data (typically bytes) to serial signals and back.

The gumstix PXA processor contains four UARTs, two of which are accessible via four-pin headers on the robostix board. The first accessible UART (FFUART) is typically controlled by the Linux console. The second accessible UART (STUART) is available for general purpose use, including its use in programming the AVR microcontroller. The AVR itself also features two UARTS. The USB serial interface connector also contains a UART which is accessible over USB by a host computer.

4.2 Host Communications Setup

To communicate with the gumstix console, the host PC must run terminal communication software such as minicom or HyperTerminal. Terminal communication software was once popularly used in conjunction with modems for text-mode access to computer Bulletin Board Systems (BBSes) and remote Unix systems.[1] Connecting to the gumstix with terminal software is exactly like connecting to a remote Unix system, except that there's no modems or phone lines involved.

To communicate with the gumstix, the terminal software must be configured to use the USB serial device at 115200 bps with 8 data bits, no parity, 1 stop bit, and no flow control (neither hardware (RTS/CTS) nor software (XON/XOFF)). The following sections provide step-by-step guides for configuring these settings in popular terminal software packages.

4.2.1 Configuring Minicom on Linux

To configure minicom for connecting to the gumstix console on Linux:

1. Execute "`minicom -sc on`" to bring up the configuration menu.
2. Make the following changes under the "Serial port setup" menu:
   - Change "Serial Device" to "/dev/ttyUSB0"[2]
   - Change "Bps/Par/Bits" to 115200 8N1.
   - Change "Hardware Flow Control" to "No".
   - Change "Software Flow Control" to "No".
3. Make the following changes under the "Modem and dialing" menu:
   - Clear the "Init string" field.
   - Clear the "Reset string" field.
   - Clear the "Hang-up string" field.

[1] With the rise in popularity of the Internet, text-mode terminal connections were replaced by SLIP and PPP to enable the connecting machine to participate in the IP network directly—this is what's now known as "dialup Internet."

[2] Or the appropriate device if not ttyUSB0.

4. Select "Save setup as dfl".
5. Select "Exit from Minicom".

To invoke minicom for normal operation:

- Execute "`minicom -c on`" to enter minicom.
- Type "`C-A q`"[3] to exit minicom.

4.2.2 Configuring HyperTerminal on Windows XP

To configure HyperTerminal for connecting to the gumstix console on Windows XP:

1. Execute HyperTerminal from the Start menu (located in Accessories, Communications).
2. Cancel the "Location Information" dialog every time it appears.
3. Enter "gumstix" for the connection name, click OK.
4. Select the highest COM port under "Connect using",[4] click OK.
5. Select these settings on the "COM Properties" dialog:

Bits per second:	115200
Data bits:	8
Parity:	None
Stop bits:	1
Flow control:	None

6. Exit HyperTerminal, save the connection settings.

To invoke HyperTerminal for normal operation, execute HyperTerminal via `gumstix.ht` from the Start menu (located in Accessories, Communications, HyperTerminal).

4.3 Configuring HyperTerminal on Windows Vista

Hyperterminal is not available in Windows Vista by default. You can acquire Hyperterminal by copying the executable and dlls from a Windows XP machine. For your convenience, the relevant files are compressed and placed on Blackboard (look at the Resources tab). Simply extract the contents of the archive and double click on `hyperterm.exe` to run Hyperterminal.

When you first connect the brainstem to your Windows Vista machine, a pop-up bubble should notify you of the COM port alloted to the brainstem. Use this COM port when configuring Hyperterminal. You can now use the instructions in section 4.2.2 to configure your Hyperterminal.

If you are having problems connecting with HyperTerminal, it may be because Windows has not have configured the brainstem correctly. Go to the Windows "Device Manager", right-click and select the properties option for the COM port. Check that the Baud rate/Data bit/Parity/Stop bit/flow control etc. are as stated in Section 4.2.

4.4 Transferring Files

An RS–232 serial connection, much like Ethernet, is inherently unreliable. Thus, weak serial transmissions occasionally garble or even drop characters. While corruption is rare enough that it usually does not hinder one's ability to read text, corruption is deleterious to transferring binary data.

Much as TCP was designed to provide reliable communication streams in unreliable network environments, numerous file transfer protocols[5] have been developed to enable reliable file transfers over serial connections. For transferring files with the gumstix, we recommend using the ZMODEM protocol as it is both fast and easy to use.

[3] "`C-A`" means "press & hold the `control` key while pressing the `a` key."
[4] Or the appropriate device if not the highest COM port.
[5] Kermit & ZMODEM are the two most popular protocols. XMODEM & YMODEM were once popular but are now deprecated.

4.4.1 Transferring Files via ZMODEM in Minicom

To upload a single file from the host PC to the gumstix:

1. Make sure the gumstix console is sitting at a shell prompt. The `rz` program will be invoked automatically by minicom.
2. Type "`C-A s`" to bring up the "Upload" menu and select "zmodem".
3. Navigate to the appropriate location, pressing `SPACE` twice to enter a directory.
4. Tag the appropriate file by pressing `SPACE` once, then press `ENTER` to begin the transfer.

To download a single file from the gumstix to the host PC:

1. Execute "`lsz -b filename`" to initiate a ZMODEM transfer for the file "filename".
2. Minicom will automatically start downloading the file.

4.4.2 Transferring Files via ZMODEM in HyperTerminal

To upload a single file from the host PC to the gumstix:

1. Make sure the gumstix console is sitting at a shell prompt. The `rz` program will be invoked automatically by HyperTerminal.
2. Select "Send File" from the "Transfer" menu.
3. Select the appropriate file to upload, choose "Zmodem with Crash Recovery" under "Protocol", click Send. lsz

To download a single file from the gumstix to the host PC:

1. Execute "`lsz -b filename`" to initiate a ZMODEM transfer for the file "filename".
2. HyperTerminal will automatically start downloading the file.

4.4.3 Transferring Multiple Files

Minicom and `lsz` both allow multiple files to be transmitted via ZMODEM in the same invocation. However, since the serial link is slow and ZMODEM uses no compression internally, transferring many files may be a time consuming process.

An alternative for transferring multiple files is to create a compressed archive of the directories or files desired to transmit and transmit the compressed archive instead (see Section 5.4 for details on using tar to create compressed archives).

5 Gumstix Programming Environment

5.1 Introducing the Gumstix

Once the gumstix hardware is setup and terminal communication software is setup and running on the host, connect the gumstix power adapter to power on and boot the gumstix.

When the gumstix boots, the first text that appears on the serial console reads:

```
U-Boot 1.1.4 (Nov  6 2006 - 11:20:03) - 400 MHz - 1161

*** Welcome to Gumstix ***

U-Boot code: A3F00000 -> A3F25DE4  BSS: -> A3F5AF00
```

```
RAM Configuration:
Bank #0: a0000000 64 MB
Flash: 16 MB
Using default environment
```

U-Boot [5] is the gumstix bootloader. It is a very popular bootloader used in many 32-bit embedded systems, and is responsible for the initial hardware setup before passing control to the kernel. U-Boot will also be one of the target environments for programming in later labs.

As U-Boot proceeds, it detects a file system and Linux kernel on the MMC/microSD cards. U-Boot loads the kernel into memory, then transfers control to the Linux kernel itself. The Linux kernel initializes the remaining peripheral hardware and launches the userspace `init` process. Eventually when userspace initialization is complete, you're greeted with the login prompt on the serial console:

```
Welcome to the Gumstix Linux Distribution!

gumstix login:
```

Let's login:

1. Enter "`root`" as the login name.
2. Enter "`gumstix`" as the password.

This will bring you to a Bash shell prompt:

```
Welcome to the Gumstix Linux Distribution!

gumstix login: root
Password:
Welcome to Gumstix!
[root@gumstix ~]#
```

In the following sections, the ("`#`") prompt indicates commands that should be typed, and the non-"`#`" statements that follow indicate program output (or logical equivalent) that will be displayed as result.

5.2 Personalizing the Gumstix

Each 18–342 gumstix comes preinstalled with the same root filesystem (rootfs). After logging in, the first thing we ask you and your lab group to do is to personalize your board. At minimum, you should give your board a unique hostname (so that it is easily discernible in a Bluetooth scan), and a unique root password (so nobody can login). Follow these commands to set both:

```
# echo "choose_a_new_hostname" > /etc/hostname
# busybox hostname -F /etc/hostname
# passwd
Changing password for root
Enter the new password (minimum of 5, maximum of 8 characters)
Please use a combination of upper and lower case letters and numbers.
Enter new password:
Re-enter new password:
Password changed.
```

5.3 Shell & Environment

When you login to the gumstix as root, the shell starts with a working directory of `/root`.[6] We recommend that all work performed on the gumstix be located in `/root` or a subdirectory thereof.

[6] This may be verified by executing "`pwd`".

Installed on the MMC rootfs is the full set of GNU Coreutils (`cat`, `cp`, `echo`, `ln`, `mkdir`, `mv`, `pwd`, `rm`, `rmdir`, etc.) [9] and other GNU utilities (`find`, `grep`, etc.). Although we do not expect you to be an expert, you should have some familiarity with the these tools as they will be essential for working in the gumstix environment. If you are unfamiliar with them, please consider trying a GNU/Linux tutorial.

To reboot the gumstix, execute the command "`reboot`". To halt or power off the gumstix, execute:

```
# halt
...
The system is going down NOW !!
Sending SIGTERM to all processes.
The system is halted.
System halted.
```

Once the gumstix reports "`System halted.`" it is safe to power off the gumstix. If you need to hard reboot the gumstix, simply power off the gumstix normally, then power it back on.[7]

5.4 Tar & Compressed Archives

Tar [11] is the traditional Unix utility for creating file archives. In the context of the gumstix platform, compressed file archives are useful for transferring multiple files over a slow serial connection.

Tar is typically used to compress a directory including all files and subdirectories within. Since the archives produced by tar are uncompressed, various file compression utilities are often used in tandem with tar. These compression utilities typically can only compress and decompress a single file, hence the need to pair them with an archiver. The combined output of the archiver and compressor is a compressed archive somewhat analogous to the popular ZIP archive format.

While tar itself is nearly ubiquitous[8], each of the various compression utilities[9] have various benefits and tradeoffs in terms of compression ratio, speed, complexity, and patent restrictions.[10] The most popular Unix compressor is gzip [16], which offers median performance in terms of both compression ratio and encoding/decoding speed.[11]

To create a tar + gzip (.tar.gz) compressed archive of the example directory `foo` and all its contents, execute the commands:

```
# tar cvf foo.tar foo
(list of files added to archive)
...
(creates foo.tar)
# gzip -9 foo.tar
(creates foo.tar.gz)
```

Due to gzip's popularity as a compressor for tar archives, many versions of tar include an option to call gzip internally, allowing the entire compressed archive to be created in one command:

```
# tar cvzf foo.tar.gz foo
```

The now completed .tar.gz archive may be transferred to the host PC over ZMODEM.

When ZMODEM is used to send a .tar.gz archive from the host PC, the original directory structure may be extracted from the archive with the command:

```
# tar xvzf foo.tar.gz
(creates directory foo/ and lists files extracted)
...
```

[7] There is a reset switch on the underside of the basix board. However, it is extremely inconveniently placed when connected to the robostix. Don't waste your time with it.

[8] An occasionally used alternative is cpio.

[9] Includes gzip, bzip2, lzop, lzma, compress, and others.

[10] See LZW algorithm & Unisys patent debacle.

[11] Uses the same DEFLATE compression algorithm as the ZIP format.

5.5 Coding & Editors

The course staff recommends that most of your group's coding be done on a host machine and not on the gumstix itself. Due to the fragile nature of the MMC/microSD cards, any significant code changes made on the MMC/microSD cards could be lost before you have a chance to transfer and backup.

Given the nature of the typical "edit, compile, debug" cycle, it is practical to make minor code changes on the gumstix itself while debugging. However, be sure not to leave any code on the MMC that can't be rewritten or remodified in case of loss.

For when you do need to edit code on the gumstix, there are three editors installed on the MMC rootfs: GNU nano, JED, and Vim. Here is a brief description of each:

GNU nano. nano [20] is a clone of Pico (of Pine fame) and is the simplest of the three editors. nano's feature set is, however, limited in comparison to the other two. To use nano, execute "`nano`".[12]

JED. JED [4] is a powerful but friendly programmer's editor. JED emulates Emacs by default, and is a reasonable choice for those familiar with Emacs.[13] To use JED, execute "`jed`".

Vim. Vim [18] is a very powerful vi-like editor. Although quite popular with Unix programmers, it is often regarded as unintuitive and as having a steep learning curve. Unless you're already proficient in Vim or want to commit to learning it, it's probably best to avoid in favor of JED. To use Vim, execute "`vim`".[14]

5.6 Compilers & Development

Traditionally, code for an embedded system is compiled on a desktop or workstation computer with a cross compiler.[15] To maximize lab consistency and to minimize the software requirements of the host PC, 18–342 labs will actually be compiled on the gumstix hardware itself.

The MMC/microSD rootfs features two compilers, a GCC ARM native compiler (`gcc`) [8], and a GCC AVR cross compiler (`avr-gcc`). In addition, the MMC rootfs contains both AVR Libc [21], and (ARM Linux) uClibc [1], as well as other traditional Unix development tools such as diff/patch and GNU make.

6 Gumstix Programming Exercises

The following programming exercises are meant to serve as a tutorial introduction to writing, compiling, executing, and debugging code on the ARM platform. While the code is simple, the procedural exercise will be invaluable in preparing for later labs.

6.1 Hello World on ARM (20 points)

Complete these steps to write a "Hello world!" application on ARM:

1. Boot the gumstix and login as `root`.
2. Create a `/root/lab0/hello` project directory:

   ```
   # mkdir -p ~/lab0/hello
   # cd ~/lab0/hello
   ```

3. Using the editor of your choice, edit the file `hello.c`.
4. In `hello.c`, write a version of the canonical "Hello world!" program that:

[12] If the screen appears garbled, try executing "`TERM=vt100`" first.
[13] Emacs itself could not be included due to its size and consistent refusal to cross compile.
[14] If you didn't follow the advice given, and are now stuck in Vim, type "`:q!`" to quit.
[15] A cross compiler generates code for a target platform that is different from the host platform on which the compiler executes.

- Writes the string "`Hello world!`" followed by a new line to *stdout*.
- Terminates with exit status 0.
- Style and comment your code properly.

5. Compile `hello.c` by executing the commands below, and verify that your output is proper:

```
# gcc -Wall -Werror -o hello hello.c
# ./hello
Hello world!
```

6.2 Goodbye World on ARM (20 points)

Complete these steps to write a "Goodbye world!" application on ARM:

1. Generate the assembly code for the `hello.c` program:

```
# gcc -S -Wall -Werror hello.c
(creates hello.s)
```

2. Create a new `/root/lab0/goodbye` project directory.
3. Copy the `hello.s` assembly code to the new project directory and rename it `goodbye.s`.
4. Modify `goodbye.s` assembly code so that the new behavior of the program is:
   - Writes the string "`Hello world!`" followed by a new line to *stdout*.
   - Writes the string "`Goodbye world!`" followed by a new line to *stdout*.
   - Terminates with exit status 42.
   - Comment your code properly.
5. Assemble `goodbye.s` by executing the commands below, and verify that your output matches:

```
# gcc -o goodbye goodbye.s
# ./goodbye
Hello world!
Goodbye world!
```

6.3 Disassembly (20 points)

Complete the following steps to disassemble the "Hello world!" application:

1. Change back to the `/root/lab0/hello` project directory.
2. Disassemble the `hello` executable with the following commands:

```
# objdump -d hello > hello-d.txt
# objdump -D hello > hello-D.txt
```

3. Transfer `hello-d.txt` & `hello-D.txt` to the host machine and analyze them.

Each question is worth five points. Record your answers to the following questions in a plain text file called `disassembly.txt`:

1. What is the entry point address of the program? (Hint: The `readelf` program may provide a clue.)
2. What is the name of the first function branched to in the program? (Hint: One of `readelf -s`, `readelf -r`, `objdump -t`, or `objdump -T` may provide a clue.)
3. What is the *key* difference between the output of `objdump -d` (`hello-d.txt`) and `objdump -D` (`hello-D.txt`)?
4. Is the interpretation of the instructions under the `.rodata` section of `hello-D.txt` correct? What does this interpretation mean?

7 Automating the Build Process with Make (20 points)

Manually rebuilding executables as part of the "edit, compile, debug" cycle quickly becomes a time consuming and tedious task, especially for large projects with many source files. The traditional Unix development utility used to automate and manage the build process is Make [10].

Projects using Make ship with a `Makefile` which describes how to build various "targets" of the project from source files. When Make is invoked as "`make`", it builds the first target listed in the Makefile, which is typically the project executable in its default configuration. Most projects contain other useful targets such as "`make clean`" which typically removes all temporary object files used in the building process.

7.1 Makefiles at a Glance

A `Makefile` is a plain text file that contains a set of rules. Each rule describes how to generate a target file from a list of prerequisite files and a list of shell commands. Makefile rules have the form:[16]

```
target: prerequisites ...
        commands to build target from prerequisites
        ...
```

If one of the prerequisite files specified by a rule doesn't exist, Make attempts to build that prerequisite from another rule that specifies the prerequisite as its target. Once all prerequisites are satisfied, Make builds the target by executing the build commands.

Once a target is built, it will not be rebuilt by subsequent invocations of Make unless a prerequisite is modified (and, thus, making the target out of date). This feature enables Make to automatically rebuild the minimum number of files to generate an up-to-date target, speeding up the compile portion of the "edit, compile, debug" cycle.

7.2 Example Makefiles

A simple example `Makefile` that is sufficient for building the "Hello world!" application:

```
hello: hello.c
        gcc -Wall -Werror -o hello hello.c
```

To build the hello executable, execute "`make hello`" or even "`make`".

C programs that consist of multiple source files are typically built in multiple stages. First, each source file is compiled into a separate object file, and second, each object file is linked to form the final executable. Another simple example `Makefile` that demonstrates this approach is:

```
baz: foo.o bar.o
        gcc -o baz foo.o bar.o

foo.o: foo.c common.h
        gcc -c -Wall -Werror -o foo.o foo.c

bar.o: bar.c common.h
        gcc -c -Wall -Werror -o bar.o bar.c
```

One of the major advantages to separating the compile and link stages is that it minimizes the amount of rebuilding necessary to incorporate changes from a single source file. For example, if a change is made in `foo.c`, only `foo.o` and `baz` is rebuilt since `bar.o` remains unaffected by the changes. However, if a change in `common.h` is made, then all targets need to be rebuilt.

[16] Commands specified in a rule *must* be indented by a tab, and not spaces.

7.3 Additional Make Features

Make provides a number of additional features that are utilized by Makefiles for most software packages to increase flexibility[17], and reduce redundancy compared to the simple examples presented earlier. Features typically encountered are:

Variable Assignment & Substitution. Make allows a `Makefile` to assign variables with the syntax "`var = value`", and substitution of variables into rules with the syntax "`$(var)`". Variable assignments may be overridden on the "`make`" command line. For example, most Makefiles assign the `CC` variable to `gcc` as the default compiler and write compile rules using "`$(CC)`" to invoke it. If you wanted to substitute the default compiler with the `avr-gcc` cross compiler, you would execute "`make CC=avr-gcc`" instead of "`make`".

Automatic Variables. Make automatically assigns the variables `$@`, `$<`, and `$^` when evaluating commands for a rule: `$@` is assigned the file name of the target, `$<` is assigned the file name of the first prerequisite, and `$^` is assigned a string consisting of the file names of all the prerequisites with spaces between them. This feature allows the rule:

```
foo.o: foo.c common.h
        gcc -c -Wall -Werror -o foo.o foo.c
```

to be simplified to:

```
foo.o: foo.c common.h
        gcc -c -Wall -Werror -o $@ $<
```

Implicit & Pattern Rules. Make handles rules that only specify a target and prerequisites (that is, rules that *don't* specify commands for building the target) by selecting an appropriate implicit rule to use to build the target. Many implicit rules are built-in, but they may be specified manually with a pattern rule.

A pattern rule is an ordinary rule that specifies a target, prerequisite, and commands for building the target, except that the file names for the target and prerequisite contain a wildcard ("`%`") that matches at the beginning of the file name. Pattern rules define how to build files of a certain type. For example, the following pattern rule specifies how to build an object file from any C source file:

```
%.o: %.c
        gcc -c -Wall -Werror -o $@ $<
```

7.4 A Real-World Makefile

Combining the above Make features, an example of a typical real world `Makefile`[18] is:

```
CC     = gcc
CFLAGS = -O2 -Wall -Werror

objects = foo.o bar.o

default: baz

.PHONY: default clean clobber

baz: $(objects)
        $(CC) -o $@ $^

foo.o: foo.c common.h
```

[17] For example, by allowing the user to substitute for a different compiler.

[18] Or at least, a Makefile you're likely to encounter in later labs.

```
bar.o: bar.c common.h

%.o: %.c
	$(CC) -c $(CFLAGS) -o $@ $<

clean:
	rm -f $(objects)

clobber: clean
	rm -f baz
```

In general, you will not be expected to write this kind of Makefile from scratch. However, it is expected that you understand what this Makefile does, and be able to modify it to include additional source files in your own projects.

7.5 Makefile Exercise

Complete these steps to write a simple calculator program:

1. Create a `/root/lab0/calc` project directory:
2. Create a `math.c` source file containing five functions; `add`, `sub`, `mul`, `div`, and `mod`; that implement integer addition, subtraction, multiplication, division, and modulo (remainder of division) respectively. Each function should take two integer arguments and return an integer result.
3. Create a `math.h` header file that contains function prototypes for each of the five functions implemented in `math.c`.
4. Create a `calc.c` source file with a single `main` function that implements a simple calculator program with the following behavior:
   - Accepts a line of input on *stdin* of the form "`number operator number`" where `number` is a signed decimal integer, and `operator` is one of the characters "`+`", "`-`", "`*`", "`/`", or "`%`" that corresponds to the integer addition, subtraction, multiplication, division, and modulo operation respectively.
   - Performs the corresponding operation on the two input numbers using the five `math.c` functions.
   - Displays the signed decimal integer result on a separate line and loops to accept another line of input.
   - Or, for any invalid input, immediately terminates the program with exit status 0.
5. Create a `Makefile` using the above "Real-World Makefile" example[19] as a boilerplate. Modify the `Makefile` so that it builds the executable `calc` as the default target. The `Makefile` should also properly represent the dependencies of `calc.o` and `math.o` (hint: try "`gcc -MM`").
6. Compile the `calc` program by executing "`make`" and verify that the output is proper. For example:

   ```
   # ./calc
   3 + 5
   8

   6 * 7
   42
   ```

[19] `/Blackboard.andrew.cmu.edu/Labs/Lab1/Part1/example.mk`

8 Completing the Lab

8.1 What to Turn In

When finished with this part of the lab, please submit *only* the following source code and project files (maintaining project directory paths) on Blackboard (use the Digital Dropbox to submit your files). Before submitting your files, create a folder `lab1-part1-group-xx` (where xx is your group number) that contains only the files listed below. Tar and gzip the folder and upload the .tar.gz version on blackboard.

- `lab1/hello/hello.c`
- `lab1/goodbye/goodbye.s`
- `lab1/hello/hello-d.txt`
- `lab1/hello/hello-D.txt`
- `lab1/calc/math.c`
- `lab1/calc/math.h`
- `lab1/calc/calc.c`
- `lab1/calc/Makefile`

Please **also** submit your written answers to the questions asked in Section 2.6 and 6.3 in plain text format[20] to:

- `lab1/repository.txt`
- `lab1/disassembly.txt`

8.2 Where to Get Help

Documentation for most Unix utilities are available as man pages (e.g., "`man objdump`"). Due to space constraints, man pages and other documentation are *not* included on the gumstix. However, they should be readily available on any Linux machine including Andrew Linux servers and any machine in the ECE undergraduate cluster, or online [19]. Documentation on the gumstix platform is available on the gumstix wiki [15].

TAs are available to answer questions during office hours. You may also email the course staff[21] with any question. Hours & availability details are available on the course website [13].

8.3 Additional Notes

Reminder: You and your lab group are responsible for maintaining backups of source code while working on lab projects. **Do not store code on the MMC/microSD only.** Flash memory has a limited number of write cycles and wears out quickly. It is best to store project source code in Andrew or ECE AFS where it is backed up nightly.

The course staff also recommends the use of a revision control system such as Subversion [3], Mercurial [17], or Git [23] for managing source code among partners and to maintain version history. Although it is not necessary for this or future labs, use and proficiency of such a system will become valuable in later labs as code size and complexity increases.

[20] ASCII or UTF-8 encoding.

[21] use the staff group on blackboard

9 Optional Reading

None of the information in this section is required knowledge for this lab—you may stop reading here if you like. However, if you're interested in getting the most out of your gumstix, this section contains potentially useful tips.

9.1 Using Bluetooth for Host Communication

Although one may interact with the gumstix using a serial console alone, using Bluetooth allows for full networking between the host computer and the gumstix at speeds approximately eight times faster than the serial connection.

9.1.1 Bluetooth Setup on the Host

To setup Bluetooth on the host:

1. Setup a PAN profile with the host listening as a Group ad-hoc Network controller.
2. Set the following network settings for the Bluetooth network device:

IP address:	172.16.0.1
Netmask:	255.255.255.0
Network address:	172.16.0.0
Broadcast address:	172.16.0.255

9.1.2 Bluetooth Setup on the Gumstix

To setup Bluetooth on the gumstix:

1. Edit `/etc/default/bluetooth`:
   - Change "`PAND_ENABLE=false`" to "`PAND_ENABLE=true`".
   - Replace the `XX`s in "`PAND_OPTIONS`" with the host's Bluetooth device address.
2. Execute "`/etc/init.d/S30bluetooth stop; /etc/init.d/S30bluetooth start`".

Once configured, the Bluetooth `pand` should execute on boot, but doesn't always in practice. If pand isn't running, execute:

```
# /etc/init.d/S30bluetooth stop; /etc/init.d/S30bluetooth start
```

9.1.3 Communicating via Bluetooth

Once Bluetooth is setup on both the host and the gumstix, the host may communicate with the gumstix as it would with any networked device. One may shell into the gumstix with `ssh` as user `root` to address 172.16.0.2. Alternatively, one may copy files with `scp`, `sftp`, or `sshfs`.

9.2 Root Filesystems

The 18–342 gumstix kits actually come with two root filesystems. The primary rootfs is installed on the MMC and boots automatically when the MMC is present—this is the rootfs used in the labs. The secondary rootfs is installed on the onboard StrataFlash and is booted when the MMC is absent.[22] Both rootfses are essentially the same, except that the StrataFlash rootfs lacks most editors, compilers, and a few other utilities due to the StrataFlash's 16 MB space constraint.

You shouldn't need to use the StrataFlash rootfs in labs. However, be aware that it does exist in case you accidentally boot without the MMC inserted.

[22] The secondary rootfs may also be booted even with the MMC present by issuing the "`icache on; fsload && bootm`" command at the U-Boot "`GUM>`" prompt.

10 Lab 1 Part 2: Code Optimization

This part of lab 1 (henceforth called lab 1) is designed to acquaint you and your partner with the skills necessary to perform space (memory) and time (performance) optimization in embedded systems. This lab should allow you to apply theory (i.e., the material taught in the lectures on assembly-language programming, profiling and optimization) to hands-on practice. There are two sub parts to this lab.

- Part I deals with optimization practices in ARM assembly-language code.
- Part II deals with optimization in the high-level (C) language.

11 Part I: Assembly-Language Optimization (50 points)

11.1 Code Distribution

The files for Part I of the lab are located on the course Blackboard site.
You will see three files in this directory, as indicated from the output of "`ls -al`" in this directory.

```
-rw-r--r-- 1 root root 399 Sep 24 00:31 Makefile
-rw-r--r-- 1 root root 695 Sep 24 00:31 part1-main.c
-rw-r--r-- 1 root root 665 Sep 24 0o:31 part1-strTable.s
```

Of the two source files, the `.c` file is written in C and the `.s` file is written in the ARM assembly language. If you examine these files, you will notice that `main()` in the `part1-main.c` file invokes a function called `strTable()`.

The `strTable()` function is defined in a file called `part1-strTable.c` that is not provided to you. Instead, the ARM assembly-language equivalent of `part1-strTable.c` is contained in the `part1-strTable.s` file. In case you are curious, here is how we generated `part1-strTable.s` from `part1-strTable.c`.

```
[root@gumstix lab1]# gcc -S -O -fomit-frame-pointer -mcpu=xscale part1-strTable.c
```

Use the provided `Makefile` to link the two files, `part1-main.c` and `part1-strTable.s`, together to form an executable called `part1`.

11.2 Focus

Note that the `-O` option has already been used with `gcc` to generate the ARM assembly code contained in the `part1-strTable.s` file. However, there is still plenty of room for optimization, based on the principles that were taught in the 18-342 lectures.

You need to optimize only the ARM assembly code in the `part1-strTable.s` file. We are looking for you to employ optimizations that improve the performance of the `strTable()` function. You do not need to worry about any optimizations in any part of the program outside of the `strTable()` function. In fact, we expressly forbid any optimizations in `part1-main.c` (particularly since we intend to use a different `part1-main.c` file to test your optimized `strTable()` function).

We are intentionally not providing you with a high-level specification of what `strTable()` accomplishes. You can infer what the `strTable()` function does by looking through its assembly-language code.

11.3 Test Cases

Here's what you should see when you run the executable, `part1`, on the gumstix.

```
[root@gumstix lab1]# ./part1
PRE: src:  theinitialstring (16 bytes), dst:  PENGUINS (8 bytes)
```

```
POST: src:  EhGIitiaNsPriSgU (16 bytes), dst:  PENGUINS (8 bytes)
```

We recommend that you generate additional test cases of your own, to check for yourself that the (given) unoptimized and (your) optimized versions of `strTable()` program produce the same output. To generate additional test cases, you will need to edit the `part1-main.c` file to modify the `src[]` and `dst[]` strings.

After the optimizations, your code should still work correctly with all of the test cases. Thus, the optimized `strTable()` should still be **logically** equivalent to its unoptimized version.

11.4 Grading Criterion

Keep in mind that your goal in optimizing the code is to try to reduce both the runtime as well as the code size. Therefore, you will be graded on the following cost-based criteria (which by the way doesn't directly translate in actual points for this lab; it's just so we can gauge your performance):

The overall cost of your optimized version will be a combination of two things for each instruction: instruction memory cost, and instruction runtime cost. Following are the detailed costs:

Instruction memory cost:

- ALL instructions = 5pt each

Instruction runtime cost will be determined as such:

- Conditional Branch instructions = 3pts each
  Reasoning: conditional branch instructions, in most cases, will stall the pipeline of a processor; so in runtime, it could cost up to 3 cycles (assuming a 3-stage pipeline).
- Load-store instructions = 3pts each
  Reasoning: memory loads and stores are expensive and in most ARM processors, the pipeline is not designed to skip stages.
- Load-store multiple instructions = 3pts * number of registers involved
  Reasoning: if three registers are loaded together, the memory bandwidth is same as three separate loads. However, the instruction memory cost will be reduced using this.
- ALL other instructions = 1pt each

Course staff will assess your cost based on `PRE: src:  theinitialstring (16 bytes), dst:  PENGUINS (8 bytes)` input.

So, to come up with your overall cost, you should trace through your program, figure out what is being executed and what is dead code (so you can eliminate it), and look at each instruction and figure out the cost associated with it. To make your life easier, you can use GDB. The debugger may n't work on the gumstix right out-of-the-box, but just follow the instructions in the following section to get it up and running very quickly.

While doing optimizations, keep in mind that `gdb` will choke if you change the value of `lr` at any point. So we forbid you to touch `lr` or the last line in `strTable (mov pc, lr)`. We will only count `mov pc, lr` as an always branch rather than a conditional branch.

If you fail to appropriately document all versions of your strTable function and the optimization mechanisms used, you will lose points. Also, if your optimized function fails our validation tests, points will be taken off based on the severity of the error.
Example of documentation:

```
SUBS r1, r1, #1  replaced 2 instructions SUB and CMP with SUBS
```

To give you an idea on the cost of a program, we have tallied the base implementation:

Type	Count
Instructions	26
Load-Store	37
Conditional-branch	41
Normal	85

Total cost: 449

Sample implementation:

Type	Count
Instructions	21
Load-Store	28
Conditional-branch	25
Normal	83

Total cost: 347

Note: small tallying error may have occurred; these are just to give you a general idea.

With all that said, students who turn in correct, valid implementations achieving a cost below 380 will be guaranteed at least 90% of the points on this section, given that they didn't lose any style or lateness points. Course staff may send updates on this value as the assignment progresses. Extra credit points will be given to top 5-10% of teams or for other creative endeavors.

11.5 Debugging with GDB

To run `gdb`, you would do something like:
`[root@gumstix lab1]# gdb part1`
Now, at the `gdb` prompt, BEFORE attempting to run your program, enter the command:
`(gdb) set osabi default`
To avoid having to do this every time you start `gdb`, you can make this automatic by simply placing the `"set osabi default"` command in the `~/.gdbinit` file (please note that your gumstix may already have this line in this file).
To summarize, something like this should get you started on debugging your program:

```
[root@gumstix lab1]# gdb part1
(gdb) set osabi default
(gdb) break strTable
(gdb) run
```

To step through your program you can then go ahead and use the step command. There are many resources online on `gdb` usage.

11.6 Hints

As you know, loop unrolling has the potential to reduce the number of branches taken and the number of instructions executed; but at the same time, you need to keep in mind that unrolling loops too much drastically increases the amount of code. Remember that this lab is an interesting balance between the runtime performance and the amount of code space. If you get stuck optimizing from one perspective, you

can always change to the other.
Because the nature of the solution may vary and there is by no means, a ”perfect” solution, the course staff may give up to 10 points of extra credit for outstanding performance. Outstanding performance involves turning in code such that all versions are well documented and the overall performance index is within that of the top few groups in the class. We encourage you to challenge yourself and to not limit yourself by the staff’s index.

12 Part 2: High-Level Optimization (50 points)

As with any industry level software you will write in your career, it is important to optimize an algorithm to reduce both the runtime complexity and the memory required.

12.1 Oddball Optimization (50 points)

Given an array of $2n - 1$ integers containing integers between 1 and n inclusive, every integer, except one, will appear twice in the array.

Write a function that finds the integer that occurs only once. A base implementation is given to you; NO credit will be given if you turn it in as is. See the base implementation in `part2.c` function `oddball`. Assume the input array is always properly formatted and it follows the guidelines mentioned above. Also, assume $n \geq 2$.

Show *at least two ways* of optimizing this algorithm and place them both in `part2.c`. Indicate using comments about the optimizations you have made and why you made them. Label the best version of your algorithm as function `oddball`. Keep all other versions of the `oddball` function in `part2.c` either commented out (use `#ifdefs`) or label the function differently. We will grade you based on the runtime complexity and memory usage. We will deduct points for any inefficient or unnecessary code in your function. Please understand the function `oddball` before jumping to optimize it.

With the Makefile and `part2-main.c` provided, doing `make N=100`, to compile the code will cause the value of n to be set to 100.

Code provided in handout for your convenience:

```
int oddball(int *arr, int len)
{
    int i, j;
    int foundInner;
    int result = 0;
    for (i = 0; i < len; i++)
    {
        foundInner = 0;
        for (j = 0; j < len; j++)
        {
            if (i == j)
            {
                continue;
            }
            if (arr[i] == arr[j])
            {
                foundInner = 1;
            }
        }
        if (foundInner != 1)
        {
            result = arr[i];
        }
    }
    return result;
}
```

12.2 Code Distribution

The archive `part2.tar.gz` on Blackboard contains the following files

1. Makefile: `makefile` will compile your code into multiple binary files
   - `part2_def`: default binary using base implementation of oddball and randGenerator
   - `part2_o1`: oddball optimization version 1
   - `part2_o2`: oddball optimization version 2
2. `part2-main.c`: driver for the program, it is suggested you add validation code in here
3. `part2.c`: you should add your optimized code here, make sure the function signature does not change
   - Under `#ifdef OPTIMIZE1`, place your first version of `oddball` here
   - Under `#ifdef OPTIMIZE2`, place your second version of `oddball` here

12.3 Documentation

For each version of your function, write the improvements made in the function and how it is better than the previous version (memory, runtime complexity). Add a comment on top of each file with the name of all group members.

12.4 Code Validation

While grading your assignment, the course staff will run a series of validation tests on your code. Depending on the severity of the flaws in your code, you will get points deducted. It is highly recommended you write code to validate both your `randGenerator` function.

13 What to Turn In (for Part 2)

For this part of Lab 1, your group needs to create one archive: `lab1-part2-group-xx.tar.gz` (where `xx` is your group number) and upload it on Blackboard. This archive should contain

- `strTable.s` which should contain your optimized code for part 1 of this lab (part)
- `part2.c` which should contain your optimized code for `oddball` function
- `par2-main.c` which should contain any validation code you may have written to validate the optimizations for part 2 of this lab (part)

Before submitting your optimized code, you should verify that your `strTable.s` file actually compiles with the provided `part1-main.c` to produce a working executable, `part1`. We may test your optimized code by compiling your submitted `strTable.s` file with a new `part1-main.c` file where we have changed the `src[]` and `dst[]` strings to be other than the test cases we have provided you.

Although `part2-main.c` is not mandatory for submission, if you wrote validation code we would like to see it.

References

[1] E. Andersen. uclibc [online]. Available from: http://www.uclibc.org/.

[2] S. Chacon. Pro git [online]. Available from: http://git-scm.com/book.

[3] CollabNet. Subversion [online]. Available from: http://subversion.tigris.org/.

[4] J. E. Davis. The JED editor home page [online]. Available from: http://www.jedsoft.org/jed/.

[5] DENX Software Engineering. Das U-Boot: the universal boot loader [online]. Available from: http://www.denx.de/wiki/UBoot.

[6] Dick Grune. Concurrent versions system [online]. Available from: http://www.nongnu.org/cvs/.

[7] V. Driessen. A successful git branching model [online]. Available from: http://nvie.com/posts/a-successful-git-branching-model/.

[8] Free Software Foundation. Gcc, the GNU compiler collection [online]. Available from: http://gcc.gnu.org/.

[9] Free Software Foundation. GNU Coreutils [online]. Available from: http://www.gnu.org/software/coreutils/.

[10] Free Software Foundation. GNU Make [online]. Available from: http://www.gnu.org/software/make/.

[11] Free Software Foundation. GNU Tar [online]. Available from: http://www.gnu.org/software/tar/.

[12] Future Technology Devices International. Virtual COM port drivers [online]. Available from: http://www.ftdichip.com/Drivers/VCP.htm.

[13] R. Gandhi. 18–342 fundamentals of embedded systems [online]. Available from: http://www.cmu.edu/blackboard/.

[14] Github. Code school - try git [online]. Available from: http://try.github.com.

[15] gumstix. Gumstix support wiki [online]. Available from: http://docwiki.gumstix.org/.

[16] J. loup Gailly and M. Adler. The gzip home page [online]. Available from: http://www.gzip.org/.

[17] M. Mackall. Mercurial [online]. Available from: http://www.selenic.com/mercurial/.

[18] B. Moolenaar. Vim the editor [online]. Available from: http://www.vim.org/.

[19] Name.net. Linux man pages [online]. Available from: http://www.linuxmanpages.com/.

[20] Nano Core Development Team. GNU nano [online]. Available from: http://www.nano-editor.org/.

[21] T. A. Roth and J. Wunsch. AVR libc home page [online]. Available from: http://www.nongnu.org/avr-libc/.

[22] D. Roundy. Darcs [online]. Available from: http://darcs.net/.

[23] L. Torvalds and J. C. Hamano. Git: Fast version control system [online]. Available from: http://git-scm.com/.