## 1 Introduction

**Bit-level operations** are common in C and systems programming. This assignment features a problem in which shifting and bitwise AND/OR-ing are required to complete the requirements.

Debugging is also a critical skill enabled by the **debugger**. The second problem in the assignment makes use of the GNU Debugger, gdb, to work through a puzzle program requiring specific inputs to pass its "phases".

### 1.1 Makefile

As in the first assignment, a Makefile is provided as part of this project. The essential targets pertinent to each problem are described in those sections.

### 1.2 Automated Tests

As in previous assignments, automated tests are provided and associated with problems. Each problem describes how to run tests associated with it.

## 2 Download Code and Setup

Download the code pack linked at the top of the page. Unzip this which will create a project folder. Create new files in this folder. Ultimately you will re-zip this folder to submit it.

| **File** | **State** | **Notes** |
| --- | --- | --- |
| Makefile | Provided | Build file to compile all programs |
| lcd.h | Provided | Problem 1 header file |
| lcd\_main.c | Provided | Problem 1 main() function |
| lcd\_clock.c | Provided | Problem 1 lcd clock drawing functions |
| lcd\_update.c | Create | Problem 1 functions to write |
|  |  |  |
| test\_lcd.c | Testing | Problem 1 binary tests for lcd\_upate.c |
| test\_lcd\_main.sh | Testing | Problem 1 shell tests for lcd clock |
| test\_lcd\_main\_data.sh | Testing | Problem 1 shell test data for lcd clock |
| puzzlebox.c | Provided | Problem 2 Debugging problem |
| input.txt | Edit | Problem 2 Input for puzzlebox, fill this in |

## 3 Problem 1: LCD Clock Simulation

### 3.1 Overview: Delta Arrays

You are tasked with writing code which will be run by a microcontroller in a digital clock. The hardware has the following relevant features.

* Time of Day Register
* LCD Display Port Register
* User code to update the display
* A simulator program with Makefile to test your code

Each feature is discussed in subsequent sections.

#### Time of Day Register

A special register is periodically updated to contain an integer which is the number of seconds since the beginning of the day. This special register is accessible in C programs via a global variable called

int TIME\_OF\_DAY\_SEC;

You do not need to define this variable as it is already there. You do not need to set this variable as it is automatically changed by the hardware. Instead, you will need to access its value to determine various aspects of the current time relevant to display.

* Whether it is AM or PM (AM is hours 12midnight to 11:59am, PM if 12noon to 11:59pm)
* The hour of the day (12-hour format so this is 1-12)
* The ones and tens digits in the hour (blank or 1 for tens, 0-9 for ones)
* The time in minutes (0-59)
* The ones and tens digits in the minutes (0-5 for tens, 0-9 for ones)

#### LCD Display Port Register

A special register controls the display of the LCD clock. This register is accessible in C programs via a global variable called

int LCD\_DISPLAY\_PORT;

Your code can examine the current values of the register but this is not relevant to the present problem. More importantly you will need to set the bits of this variable to properly show the time.

The following diagram shows bit patterns for various digits and how they will be displayed in the ones digit of the minutes place. Digits are displayed by darkening certain **bars** in the display which correspond to certain bits in the LCD\_DISPLAY\_PORT register being set.

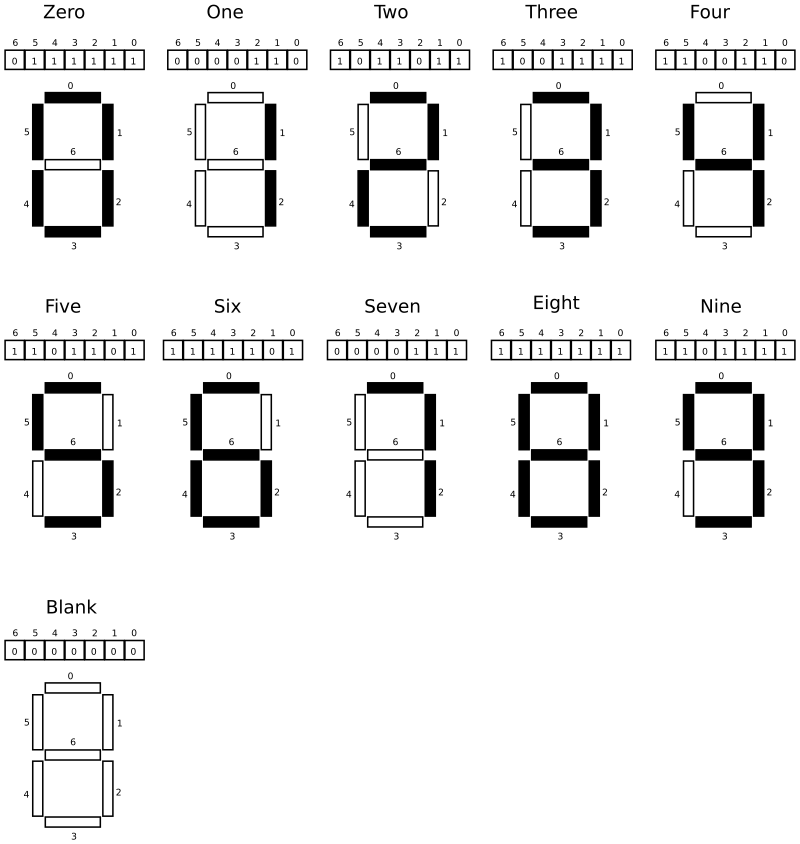


Figure 1: Bit to LCD clock display bar correspondence for the ones digit of the minute. The 0th bit controls the upper horizontal bar, the 1th bit controls the horizontal bar clockwise from it, and so on around the 6 outer bars in a clockwise fashion. The 6th bit controls the middle horizontal bar. When a bit is 1 (set), the bar will be darkened while when the bit is 0 (clear) the bar will not be displayed (shown as empty). The combinations of bits shown are the only ones that arise when showing digits for times.

Notice the following.

* Bits that are set (equal to 1) will turn on (darken) one **bar** of the clock digit
* Bits that are clear (equal to 0) will turn off one bar of the digit
* 7 bits are required to control the display of one digit
* The bits are arranged with the low order bit (bit 0) at the top and progress clockwise around the digit.
  + Bit 0 top
  + Bit 1 upper right
  + Bit 2 lower right
  + Bit 3 bottom
  + Bit 4 lower left
  + Bit 5 upper left
  + Bit 6 middle
* The programmer can set bits to any pattern which will be displayed but only patterns below correspond to digits of interest.

Time is displayed with several adjacent digits along with an AM/PM display. The diagram below shows two full times along with the bits representing the digits. The bits correspond to how the global variable LCD\_DISPLAY\_PORT should be set in order to make the clock appear as it does.

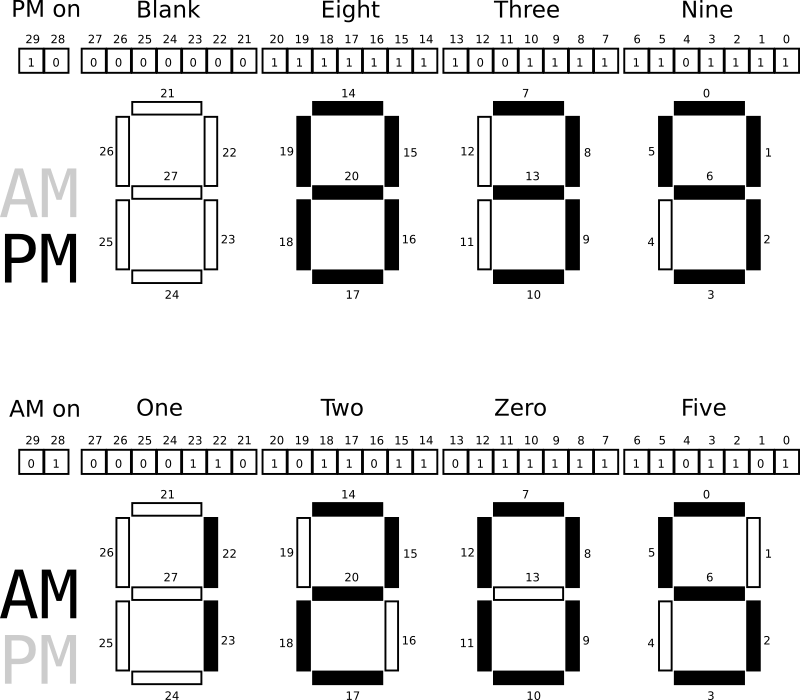


Figure 2: Two full examples of how the 30 bits of the clock display state control which parts of the clock are shown. Each digit follows the same pattern of bit to bar correspondence as the right-most with bits. The lowest order (rightmost) bit controls the top bar of each digit and proceed around the outside clockwise with the final bit for each digit controlling the middle bar. The highest order bits (28 and 29) control whether the "AM" or "PM" lights are shown. Note that both could be shown at the same time or neither shown but this should not be done for actual times.

Notice the following.

* You may presume that the LCD\_DISPLAY\_PORT register is a 32-bit integer.
* 30 bits are used to control the full clock display.
* Bits 0-6 control the ones place of the minutes
* Bits 7-13 control the tens place of the minutes
* Bits 14-20 control the ones place of the hours
* Bits 21-27 control the tens place of the hours
* Bit 28 controls whether AM is displayed
* Bit 29 controls whether PM is displayed
* Bits 30 and 31 are not used and should always be 0.

### 3.2 lcd\_update.c: Updating the LCD with User Code

Periodically the microcontroller will run code to adjust the LCD display to show the current time. This function is

int lcd\_update();

and it will be your job to write this function

Rather than write everything that needs to be done within lcd\_update(), several helper functions will be used to divide this task into several more manageable and testable chunks.

These should all be written in lcd\_update.c and are as follows.

#### Converting time of Day in Seconds to a Struct

int set\_tod\_from\_secs(int time\_of\_day\_sec, tod\_t \*tod);

// Accepts time of day in seconds as an argument and modifies the

// struct pointed at by tod to fill in its hours, minutes,

// etc. fields. If time\_of\_day\_sec is invalid (negative or larger

// than the number of seconds in a day) does not tod and returns 1 to

// indicate an error. Otherwise returns 0 to indicate success. This

// function DOES NOT modify any global variables

This function works with the struct tod\_t defined in lcd.h which has the following layout.

// Breaks time down into 12-hour format

typedef struct{

short hours;

short minutes;

short seconds;

char ispm;

} tod\_t;

The process of filling in values is simply a matter of doing some division/modulo and assigning values.

#### Setting bits in an integer according to a tod\_t

int set\_display\_bits\_from\_tod(tod\_t tod, int \*display);

// Accepts a tod and alters the bits in the int pointed at by display

// to reflect how the LCD clock should appear. If any fields of tod

// are negative or too large (e.g. bigger than 12 for hours, bigger

// than 59 for min/sec), no change is made to display and 1 is

// returned to indicate an error. Otherwise returns 0 to indicate

// success. This function DOES NOT modify any global variables

//

// May make use of an array of bit masks corresponding to the pattern

// for each digit of the clock to make the task easier.

This function will need to do bit shifting along with bitwise operations to construct the correct bit patter for the clock display.

A good trick to use is to create a series of bit patterns that correspond to the various digits. For example, according to the diagrams above, the bit patter for 9 is 0b1101111. If a 9 should appear on the clock somewhere, this bit pattern should be shifted and combined with the existing bits in display so that a 9 will show. Creating similar constant *mask* patterns for each digit and AM/PM is a good way to simplify this problem.

A detailed explanation of one approach to the problem:

* Create an array of bit masks for each of the digits 0-9. The 0th element of the array contains a bit mask like 0b0111111 which represents the bits that should be set for a 0 digit, the 1th element of this array has a mask like 0b0000110 which are the bits to be set for a 1. There should be ten entries in this array in indices 0-9.
* Use modulo to determine the integer value for the ones and tens digits for both hours and minutes. Call these variables something like min\_ones and min\_tensand similarly for hours. Each variable should be in the range 0-9.
* Start with a state variable of 0 (all 0 bits).
* Use min\_ones to index into your array of masks to determine the bits that should be set for it. Combine the state variable with min\_ones mask.
* Combining bits here is a logical operation of setting all bits that are one in the mask to 1 in the state variable.
* Use min\_tens to index into your array of masks for the right mask for that digit. The bits corresponding to the tens place of the minutes is shifted to the left by 7 bits so shift the mask to the left and combine it with the state variable.
* Repeat this process for the ones digit of the hours (shifted by 14 to the left) and the tens digit of the hour (shifted by 21).
* The tens digit of the hour is special in that it should be either 1 or blank (don't show a 0 for hours 1-9) so adjust your mask appropriately before shifting.
* Set the 28th bit of the state if the time is in the AM or the 29th bit if time is in the PM.
* The state variable should now be populated.

#### Changing the clock display

int lcd\_update();

// Examines the TIME\_OF\_DAY\_SEC global variable to determine hour,

// minute, and am/pm. Sets the global variable LCD\_DISPLAY\_PORT bits

// to show the proper time. If TIME\_OF\_DAY\_SEC appears to be in error

// (to large/small) makes no change to LCD\_DISPLAY\_PORT and returns 1

// to indicate an error. Otherwise returns 0 to indicate success.

//

// Makes use of the set\_tod\_from\_secs() and

// set\_display\_bits\_from\_tod() functions.

This function makes use of the previous two functions and the global variables that correspond to the clock hardware to alter the display. It should be relatively short by making use of the previous functions.

### 3.3 LCD Clock Simulator

While we do not have actual hardware with the features mentioned, a simulator for the clock system is in the provided files lcd\_main.c and lcd\_clock.c. You do not need to modify or understand code in either file to complete the HW though it will certainly expand you C skills to spend some time examining them.

The main() function in lcd\_main.c accepts a command line argument which is a number of seconds since the beginning of the day and will call your functions for this problem and show results for it. You are encouraged to use this function to test your code incrementally

* Examine whether set\_tod\_from\_secs() is correct based on the first part of output in lcd\_main.c
* Once set\_tod\_from\_secs() is complete, examine whether the output of set\_display\_bits\_from\_tod() is correct based on the latter part of the output.
* Once both these functions are correct, examine whether lcd\_update() is correct based on the final part of the output of the main() function.

Note that there are a variety of functions in the file lcd\_clock.c which are used to simulate how the clock will display. This is also where the global variables LCD\_DISPLAY\_PORT and TIME\_OF\_DAY\_SEC are defined. However, you do not need to modify or even understand the code in lcd\_clock.c. It is only used to show how the clock would look when the LCD\_DISPLAY\_PORT bits are set.

### 3.4 Sample Runs of lcd\_main

Below are samples generated by compiling and running the main() function in the lcd\_main.c file. The code is compiled by using the provided Makefile to create the lcd\_main program. It compiles the lcd\_clock.c library along with the functions you write in the file lcd\_update.c and the main() in lcd\_main.c.

> make lcd\_main

make: 'lcd\_main' is up to date.

> ./lcd\_main 0

TIME\_OF\_DAY\_SEC set to: 0

set\_tod\_from\_secs( 0, &tod );

tod is {

.hours = 12

.minutes = 0

.seconds = 0

.ispm = 0

}

Simulated time is: 12 : 00 : 00 am

Checking results for display bits

set\_display\_bits\_from\_tod(tod, &state);

state is:

3 2 1 0

index: 10987654321098765432109876543210

bits: 00010000110101101101111110111111

guide: | | | | |

Running lcd\_update()

LCD\_DISPLAY\_PORT is:

3 2 1 0

index: 10987654321098765432109876543210

bits: 00010000110101101101111110111111

guide: | | | | |

LCD Clock Display:

~~ ~~ ~~

| |o| | | |

~~

AM | | o| | | |

~~ ~~ ~~

> ./lcd\_main 101

TIME\_OF\_DAY\_SEC set to: 101

set\_tod\_from\_secs( 101, &tod );

tod is {

.hours = 12

.minutes = 1

.seconds = 41

.ispm = 0

}

Simulated time is: 12 : 01 : 41 am

Checking results for display bits

set\_display\_bits\_from\_tod(tod, &state);

state is:

3 2 1 0

index: 10987654321098765432109876543210

bits: 00010000110101101101111110000110

guide: | | | | |

Running lcd\_update()

LCD\_DISPLAY\_PORT is:

3 2 1 0

index: 10987654321098765432109876543210

bits: 00010000110101101101111110000110

guide: | | | | |

LCD Clock Display:

~~ ~~

| |o| | |

~~

AM | | o| | |

~~ ~~

> ./lcd\_main 4170

TIME\_OF\_DAY\_SEC set to: 4170

set\_tod\_from\_secs( 4170, &tod );

tod is {

.hours = 1

.minutes = 9

.seconds = 30

.ispm = 0

}

Simulated time is: 01 : 09 : 30 am

Checking results for display bits

set\_display\_bits\_from\_tod(tod, &state);

state is:

3 2 1 0

index: 10987654321098765432109876543210

bits: 00010000000000011001111111101111

guide: | | | | |

Running lcd\_update()

LCD\_DISPLAY\_PORT is:

3 2 1 0

index: 10987654321098765432109876543210

bits: 00010000000000011001111111101111

guide: | | | | |

LCD Clock Display:

~~ ~~

|o| | | |

~~

AM |o| | |

~~ ~~

> ./lcd\_main 43199

TIME\_OF\_DAY\_SEC set to: 43199

set\_tod\_from\_secs( 43199, &tod );

tod is {

.hours = 11

.minutes = 59

.seconds = 59

.ispm = 0

}

Simulated time is: 11 : 59 : 59 am

Checking results for display bits

set\_display\_bits\_from\_tod(tod, &state);

state is:

3 2 1 0

index: 10987654321098765432109876543210

bits: 00010000110000011011011011101111

guide: | | | | |

Running lcd\_update()

LCD\_DISPLAY\_PORT is:

3 2 1 0

index: 10987654321098765432109876543210

bits: 00010000110000011011011011101111

guide: | | | | |

LCD Clock Display:

~~ ~~

| |o| | |

~~ ~~

AM | |o | |

~~ ~~

> ./lcd\_main 43200

TIME\_OF\_DAY\_SEC set to: 43200

set\_tod\_from\_secs( 43200, &tod );

tod is {

.hours = 12

.minutes = 0

.seconds = 0

.ispm = 1

}

Simulated time is: 12 : 00 : 00 pm

Checking results for display bits

set\_display\_bits\_from\_tod(tod, &state);

state is:

3 2 1 0

index: 10987654321098765432109876543210

bits: 00100000110101101101111110111111

guide: | | | | |

Running lcd\_update()

LCD\_DISPLAY\_PORT is:

3 2 1 0

index: 10987654321098765432109876543210

bits: 00100000110101101101111110111111

guide: | | | | |

LCD Clock Display:

~~ ~~ ~~

| |o| | | |

~~

| | o| | | |

PM ~~ ~~ ~~

> ./lcd\_main 47089

TIME\_OF\_DAY\_SEC set to: 47089

set\_tod\_from\_secs( 47089, &tod );

tod is {

.hours = 1

.minutes = 4

.seconds = 49

.ispm = 1

}

Simulated time is: 01 : 04 : 49 pm

Checking results for display bits

set\_display\_bits\_from\_tod(tod, &state);

state is:

3 2 1 0

index: 10987654321098765432109876543210

bits: 00100000000000011001111111100110

guide: | | | | |

Running lcd\_update()

LCD\_DISPLAY\_PORT is:

3 2 1 0

index: 10987654321098765432109876543210

bits: 00100000000000011001111111100110

guide: | | | | |

LCD Clock Display:

~~

|o| | | |

~~

|o| | |

PM ~~

> ./lcd\_main 67089

TIME\_OF\_DAY\_SEC set to: 67089

set\_tod\_from\_secs( 67089, &tod );

tod is {

.hours = 6

.minutes = 38

.seconds = 9

.ispm = 1

}

Simulated time is: 06 : 38 : 09 pm

Checking results for display bits

set\_display\_bits\_from\_tod(tod, &state);

state is:

3 2 1 0

index: 10987654321098765432109876543210

bits: 00100000000111110110011111111111

guide: | | | | |

Running lcd\_update()

LCD\_DISPLAY\_PORT is:

3 2 1 0

index: 10987654321098765432109876543210

bits: 00100000000111110110011111111111

guide: | | | | |

LCD Clock Display:

~~ ~~ ~~

| o | | |

~~ ~~ ~~

| |o | | |

PM ~~ ~~ ~~

> ./lcd\_main 86399

TIME\_OF\_DAY\_SEC set to: 86399

set\_tod\_from\_secs( 86399, &tod );

tod is {

.hours = 11

.minutes = 59

.seconds = 59

.ispm = 1

}

Simulated time is: 11 : 59 : 59 pm

Checking results for display bits

set\_display\_bits\_from\_tod(tod, &state);

state is:

3 2 1 0

index: 10987654321098765432109876543210

bits: 00100000110000011011011011101111

guide: | | | | |

Running lcd\_update()

LCD\_DISPLAY\_PORT is:

3 2 1 0

index: 10987654321098765432109876543210

bits: 00100000110000011011011011101111

guide: | | | | |

LCD Clock Display:

~~ ~~

| |o| | |

~~ ~~

| |o | |

PM ~~ ~~

### 3.5 Manual Inspection Criteria for lcd\_update.c   GRADING

| **Weight** | **Criteria** |
| --- | --- |
| 5 | set\_tod\_from\_secs() |
|  | Clear effort to do error checking of out of bounds values. |
|  | Clear flow to how each field of tod is calculated. |
|  | Correctly setting fields of tod via pointer dereference or arrow operator. |
| 5 | set\_display\_bits\_from\_tod() |
|  | Clear effort to do error checking for out of bounds values in tod parameter |
|  | Clear code that calculates digits to be displayed |
|  | Use of bit masks corresponding to digits to be displayed |
|  | Use of bitwise operators to shift bits appropriately |
|  | Use of bitwise operators to combine shifted digit bits |
|  | Clear derference/set of the integer pointed to by the display parameter |
| 5 | lcd\_update() |
|  | Use of the global variables TIME\_OF\_DAY\_SEC / LCD\_DISPLAY\_PORT |
|  | Does not re-define these variables |
|  | Use of previous two functions |
|  | Error checking on time of day calculation |

### 3.6 Tests for LCD Clock

Both binary and shell tests can be run with make test-p1

| **Weight** | **Criteria** |
| --- | --- |
| 30 | test\_lcd.c |
|  | Provides 30 tests for functions in lcd\_update.c |
|  | Compile and run using make test-p1 |
|  | 1 point per test passed |
|  | Deductions for memory problems identified by Valgrind |
| 5 | test\_lcd\_main.sh |
|  | 5 Tests of the lcd\_main which uses functions from lcd\_update.c |
|  | 1 point per test passed |
|  | Deductions for memory problems identified by Valgrind |

## 4 Problem 2: Debugging the Puzzlebox

### 4.1 Overview

The file puzzlebox.c contains source code that reads inputs from a file named on the command line. If the inputs are correct, points are awarded. If inputs are incorrect, error messages are printed.

The puzzlebox is arranged into a series of *phases* each of which has some points associated with it.

* Not all phases must be completed to get full credit but the phases must done in order.
* Each phase reads inputs from the file provided on the command line and performs calculations on them to see if they are "correct" according to various criteria
* The very first input is your **internet ID** like kauf0095 (first part of your UMN email address). This input is used to add randomness to the puzzle so that **your answers will be different from most other students**. You must you use your own internet ID.

**The purpose of this problem is get familiar with using a debugger.** This is a powerful tool that pauses programs, allows internal values to be printed and code to be stepped through line by line. It is nearly essential to use as the code in puzzlebox is intentionally convoluted in places. Being able to pause execution and print values at various points make it **much easier** to solve the puzzles.

### 4.2 input.txt Input File

Name your input file input.txt and put your internet ID in it along with some numbers like 1 2 3. Then compile and run the puzzlebox program on it.

> make puzzlebox # compile puzzlebox

gcc -Wall -g -c puzzlebox.c

gcc -Wall -g -o puzzlebox puzzlebox.o

> cat input.txt # show contents of input.txt

kauf0095 1 2 3

> ./puzzlebox input.txt # run puzzlebox with input.txt

UserID 'kauf0095' accepted: hash value = 1397510491

PHASE 1: A puzzle you say? Challenge accepted!

Ah ah ah, you didn't say the magic word...

Failure: Double debugger burger, order up!

\* Score: 0 / 50 pts \*

This is automated with the Makefile target make test-p2:

> make test-p2 # compile/run puzzlebox with input.txt

gcc -Wall -g -c puzzlebox.c

gcc -Wall -g -o puzzlebox puzzlebox.o

./puzzlebox input.txt

UserID 'kauf0095' accepted: hash value = 1397510491

PHASE 1: A puzzle you say? Challenge accepted!

Ah ah ah, you didn't say the magic word...

Failure: Double debugger burger, order up!

\* Score: 0 / 50 pts \*

These initial forays are not positive (0 / 50 points) but the real meat is in examining the source code and modifies inputs in input.txt.

### 4.3 gdb The GNU Debugger

You will definitely need to use a debugger to solve the puzzlebox and gdb is the quintessential debugger associated with our compiler gcc. It is installed by default on all lab machines and is an easy install on must Linux machines.

For a quick overview of GDB, here are some resources

* [CSCI 2021 Quick Guide to gdb: The GNU Debugger](http://www-users.cs.umn.edu/~kauffman/2021/gdb): Page describing how to start the debugger, a sample session using puzzlebox, an overview of the most common commands.
* [CppCon 2015: Greg Law " Give me 15 minutes & I'll change your view of GDB"](https://www.youtube.com/watch?v=PorfLSr3DDI): Video giving basic overview of hope to run gdb on simple programs with an emphasis on differences between "normal" mode and TUI mode
* [GNU GDB Debugger Command Cheat Sheet](http://www.yolinux.com/TUTORIALS/GDB-Commands.html): Extensive list of commands
* [Debugging with GDB](https://sourceware.org/gdb/current/onlinedocs/gdb/): Official manual for gdb

### 4.4 Typical Cycle

A typical cycle of working on puzzlebox will be the following.

* Start the debugger with puzzlebox
* gdb -tui ./puzzlebox
* Set the arguments array to input.txt
* set args input.txt
* Set a breakpoint at a phase like phase3
* break phase3
* Run the program
* run
* Do some stepping / nexting
* step
* next
* Print some variables
* print a
* print/x b
* Make some changes to input.txt in a different window
* Re-run the program to see if things have changed favorably
* kill
* run

### 4.5 Kinds of Puzzles

The puzzles presented in the different phases make use of a variety of C program techniques which we have or will discuss including.

* Bit-wise operations and their use in place of arithmetic
* String and character array manipulations
* Interpreting bits as one of several kinds of things (integer, float, character)
* More extensive C control structures like goto and labels

### 4.6 Tests for puzzlebox.c   GRADING

puzzlebox.c itself reports how many points one can earn at the end of its execution.

Currently there are **60 points available but 50 points is considered full credit.**

If any additional points are earned, they will be counted towards the overall assignment total for the course to make up for lost credit on other assignments. Additional points will NOT count as a bonus so the maximum percentage on assignments will cannot be exceeded.

## 5 Zip and Submit

### 5.1 Submit to Canvas

Once your are confident your code is working, you are ready to submit. Ensure your folder has all of the required files. Create a zip archive of your lab folder and submit it to Canvas.

On Canvas:

* Click on the *Assignments* section
* Click on the appropriate link for this lab/assignment
* Scroll down to "Attach a File"
* Click "Browse My Computer"
* Select you Zip file and press OK

### 5.2 Late Policies

You may wish to review the policy on late project submission which will cost you late tokens to submit late or credit if you run out of tokens. **No projects will be accepted more than 48 hours after the deadline.**

<http://www-users.cs.umn.edu/~kauffman/2021/syllabus.html#late-projects>