Phase 1: Basic Input and Output

Introduction

As we boot up our operating system and begin executing code, we will consider our basic architecture: CPU, RAM, and I/O. The code that we download is comprised of instructions that located in RAM and subsequently executed by the CPU. Beyond the CPU, however, we will explore interactions with external hardware components and basic input and output operations.

Table of Contents

Objectives & Outcomes

The objectives for this phase are to ensure that you can:

1. implement simple hardware drivers
2. perform hardware interactions with devices mapped to memory addresses
3. perform hardware interactions witih devices via direct input/output operations
4. output ASCII text and characters to a VGA display
5. read and decode character input from a keyboard

Requirements

Part 1: Organization

To begin phase 1, you will need to accept the project as part of your team on GitHub classroom.

Once your team and repository has been created, you will need to clone the repository individually so you can contribute and collaborate with your team members.

This repository will be used throughout the remainder of the semester for the project.

Our operating system source will be organized into the following directory structure:

`src` for source files `include` for header files

With phase 1, and each new phase, be sure to review each new or modified header file as well as reviewing new source files.

Specifically: 1. Review function headers 2. Review function definitions 3. Review comments 4. Review any definitions (i.e. `#define`)

In general, all header files will be provided and alteration is generally not permitted. If you need to create a local header file for your own purposes, you can place it within the `src` directory.

You are welcome to create new source files if needed for your implementation, but future phases may add new source file requirements, so consider that you may need to replace/rename files over time if you add your own source files if new files have names that are used by newer phases.

In phase 1, several files are provided for you:

- `Makefile`
- `include/`
    - `bit.h`
    - `io.h`
    - `kernel.h`
    - `keyboard.h`
    - `vga.h`
- `src/`
    - `bit.c`
    - `kernel.c`
    - `keyboard.c`
    - `main.`
    - `vga.c`

General Files

File	Description
Makefile	Rules to build the operating system image

Header Files

File	Description
bit.h	Bit manipulation and utilities
io.h	Hardware Input/Output functions and definitions
kernel.h	General kernel defintions and functions
keyboard.h	Keyboard definitions and functions
vga.h	VGA (Video Graphics Array) definitions and functions

Source Files

File	Description
bit.c	Implementation of bit manipulation functions
kernel.c	Implementation of general kernel functions
keyboard.c	Implementation of keyboard handling functions
main.c	Main entry point
vga.c	Implementation of VGA functionality

The provided `Makefile` will be used to build your source code and package it into a downloadable image for your SPEDE Target.

All source files will require some degree of implementation or modification. Over time you may be asked to extend or modify functionality within your source from phase to phase.

The `main` function in `main.c` will be the first code that runs and will generally be focused on initializing various components of the operating system - from hardware to data structures and variables and ultimately to executing processes that will interact with the kernel. Each new component should have an initialization routine that can/should be called.

In phase 1, however, we will be focusing on some basic functionality to perform input and output operations. These operations will require writing hardware drivers to interact with a keyboard controller for handling input and VGA hardware to facilitate output.

In addition to writing these drivers, additional functionality will be implemented to facilitate development throughout the project.

As you develop your implementations be mindful of error conditions and consder your function inputs/parameters:

- What is considered valid?
- Can you handle or correct invalid data?
- How do you handle data that is invalid?

When working with your data structures, variables, and memory, always consider boundary conditions. Reading from or writing to invalid memory locations is considered an error and could result in unexpected behavior.

Part 2: Kernel Utilities

Kernel Logging

As we develop our operating system, we will utilize several approaches to debug different behaviors and understand different situations that are occuring.

In addition to GDB for debugging complex problems, simple logging can be hepful to understand the program flow, especially leading up to a complex problem that needs the utility of a debugger.

In part one, we will implement kernel logging functions that can be used for different purposes:

- Logging errors when things go wrong
- Logging warnings (things that aren't necessarily ideal but can still proceed)
- Logging informational messages that describe important or notable details
- Logging more verbose messages for debugging
- Logging even more verbose messages to trace through a program
- Logging fatal situations that we should be very concerned about (a panic!)

These kernel logging mechanisms will be treated as separate functions that are associated with different "logging levels". The logging level is simply an integer that can be checked to supress or allow logs at higher or lower levels.

Log levels are associated with the following enumeration:

```
// List of kernel log levels in order of severity
typedef enum log_level {
    KERNEL_LOG_LEVEL_NONE,  // No Logging!
    KERNEL_LOG_LEVEL_ERROR, // Log only errors
    KERNEL_LOG_LEVEL_WARN,  // Log warnings and errors
    KERNEL_LOG_LEVEL_INFO,  // Log info, warnings, and errors
    KERNEL_LOG_LEVEL_DEBUG, // Log debug, info, warnings, and errors
    KERNEL_LOG_LEVEL_TRACE, // Log trace, debug, info, warnings, and errors
    KERNEL_LOG_LEVEL_ALL    // Log everything!
} log_level_t;
```

In our source, the variable `kernel_log_level` stores the current log level which we will default to `info`. We can implement functionality to change the log level (either changing the default value, or changing at runtime).

For fatal situations, in addition to logging, we will want to halt or stop the operation of our operating system since we know that we have encountered an unrecoverable situation. Effectively, we should `exit()` at this point. However, since we know we have a debugger at our disposal, we can implement a breakpoint on demand in this scenario to try to understand what happened.

To start you off on a path to success, one function has been written for you: `kernel_log_info`. This function accepts a single string parameter containing a string format (similar to what you would use with printf) and a variable number of additional arguments assocaited with string format operators. In effect, this function is largely a "wrapper" that leverages `vprintf()` and also prints out an additional string to indicate that this is in fact an `info` type of log message. Most importantly, this function also checks the current log level to determine if a message should be displayed or not.

For reference, the `kernel_log_info` function contains the following:

```
/**
 * Prints a kernel log message to the host with an info log level
 *
 * @param msg - string format for the message to be displayed
 * @param ... - variable arguments to pass in to the string format
 */
void kernel_log_info(char *msg, ...) {
    // Return if our log level is less than info
    if (kernel_log_level < KERNEL_LOG_LEVEL_INFO) {
        return;
    }

    // Obtain the list of variable arguments
    va_list args;

    // Indicate this is an 'info' type of message
    printf("info: ");

    // Pass the message and variable arguments to vprintf
    va_start(args, msg);
    vprintf(msg, args);
    va_end(args);

    printf("\n");
}
```

Using the above function as a reference, you must also implement the following functions:

- `kernel_log_error`
- `kernel_log_warn`
- `kernel_log_debug`
- `kernel_log_trace`
- `kernel_panic`

With `kernel_panic` there are a few special considerations:

- There is no need to check log levels: it should always log a message
- After logging the message, ensure to perform a call to `kernel_break()` to trigger a breakpoint in GDB
- After triggering the breakpoint, ensure to perform a call to `exit(1)` to ensure that the program exits

The `kernel_panic` function should log the message to both the debug console (`printf`) and also to the vga display (`vga_printf`). You may wish to customize the VGA output (although not required) to highlight that a kernel panic actually occured.

Once these functions have been implemented, use them in your operating system where appropriate. For example, when an error occurs that is unrecoverable, be sure to perform a call to `kernel_panic`. Log activity throughout your code using log levels that make sense. Log errors. Log informational steps that you would expect to see during normal operation. As you are developing and debugging, make use of debug and trace logging to get a detailed understanding of your code.

While logs may be generated throughout the code, the log level that we want to view or actually log should be configurable. The following two functions should be implemented to configure the log level specified.:

- `kernel_set_log_level` to set the log level
- `kernel_get_log_level` to retrieve the current log level

Bit Operation Utilities

Hardware interactions are a key component of this project phase. One of the important aspects of configuring or working with hardware is to understand that many hardware devices describe the configuration or status of the hardware using individual bits within a byte or other value.

To facilitate the testing, setting, clearing, or toggling of individual bits, the following functions in `bit.c` should be implemented so they *may* be used if desired:

- `bit_count`
- `bit_test`
- `bit_set`
- `bit_clear`
- `bit_toggle`

For these bit utility functions, consider this sample code that demonstrates different operations to modify bits:

```
unsigned int x = 83; // 0x53  (0101 0011)
unsigned int y = 43; // 0x2b  (0010 1011)
int z = 0;

// Bitwise AND
// 0x53 & 0x2b == 0x03  (0000 0011)
z = x & y;
printf("0x%02x = 0x%02x & 0x%02x\n", z, x, y);

// Bitwise OR
// 0x53 | 0x2b == 0x7b  (0111 1011)
z = x | y;
printf("0x%02x = 0x%02x | 0x%02x\n", z, x, y);
```

```
// Bitwise XOR
// 0x53 ^ 0x2b == 0x78  (0111 1000)
z = x ^ y;
printf("0x%02x = 0x%02x ^ 0x%02x\n", z, x, y);

// Invert (1's Complement)
// ~0x2b == 0xFFFFFFd4  (1111 1111 1111 1111 1111 1111 1101 0100)
z = ~y;
printf("0x%02x = ~0x%02x\n", z, y);

// Right Shift
// 0x53 >> 2 == 0x14    (0001 0100)
z = x >> 2;
printf("0x%02x = 0x%02x >> %d\n", z, x, 2);

// Left Shift
// 022b << 2 == 0x158   (0001 0101 1000)
z = y << 3;
printf("0x%02x = 0x%02x << %d\n", z, y, 3);
```

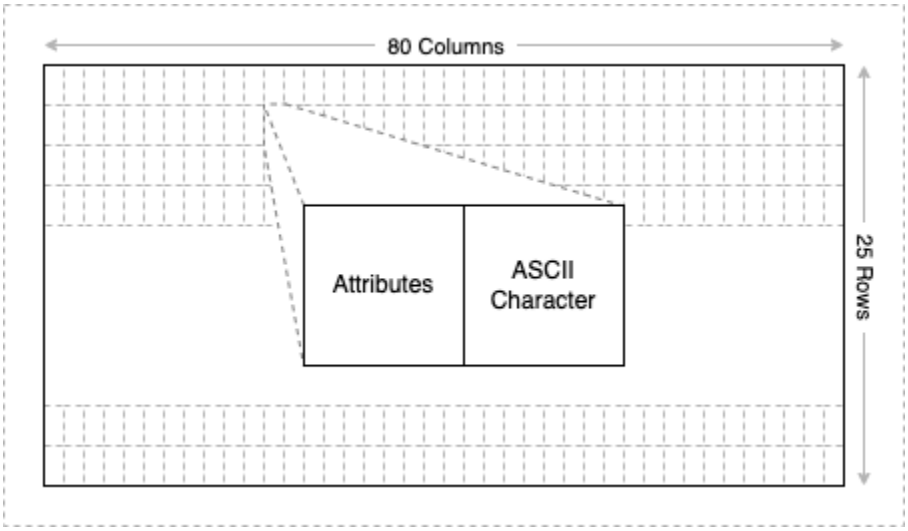
Using combinations of the above operations you can test (read) or set (write) specific bits for a given integer.

Part 3: VGA Output

In our x86 Computer System we will want to display output on the screen. To do so, we will use VGA (Video Graphics Array). VGA is a fairly old standard but is supported on all modern graphics controllers and cards. In our case, we will be starting out with using text mode (as graphics programming requires more effort) with two purposes:

1. To ensure we have a mechanism to provide output from our Operating System
2. To demonstrate an understanding of memory mapped I/O as well as some direct I/O

In our configuration, our VGA text output will support a width of 80 characters and a height of 25 characters. VGA text mode also supports attributes such as background and foreground colors, as well as supporting a cursor that can be used to indicate the current position on screen.

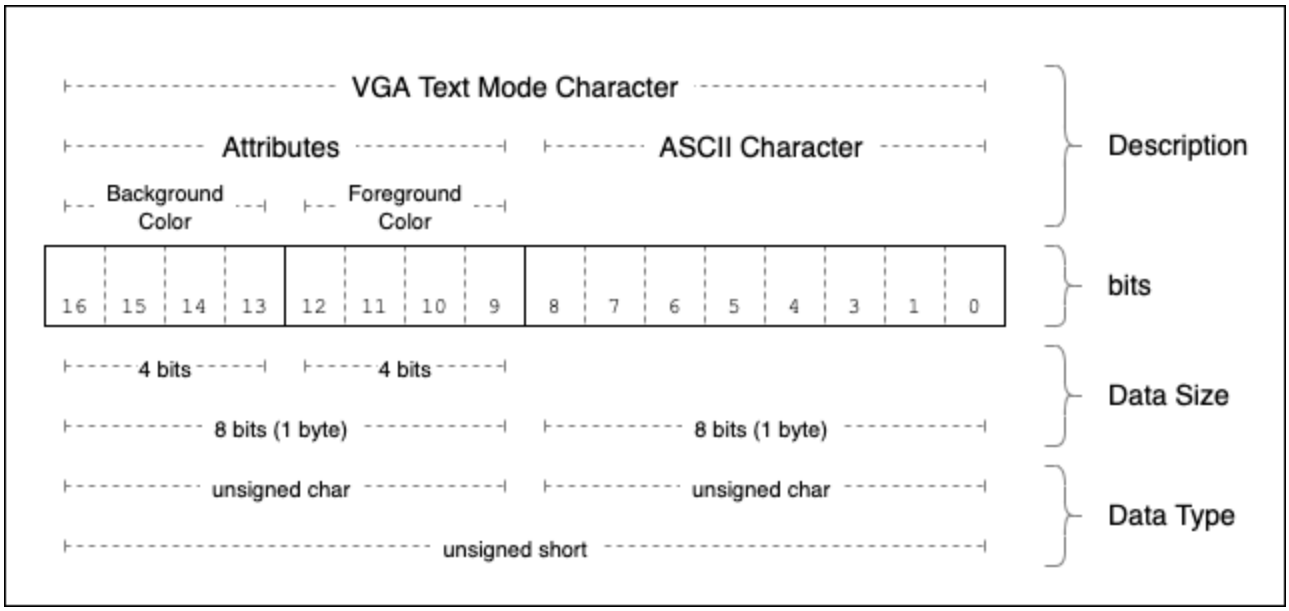


VGA text mode is programmed via Memory Mapped I/O - effecively mapping memory addresses to hardware resources. As such, a pointer to a memory address or indexing into memory makes interactions with hardware relatively easy.

In the x86 PC architecture, the memory associated with the VGA text mode output is mapped to `0xB8000`. The format of text mode data is composed of two bytes per character:

1. VGA Attributes (effectively: foreground/background)
2. ASCII Character

Each character displayed has the following byte format:



Note: Some text modes may support "blinking", in which case the 4th bit of the "background" color may be used to indicate if the character should or should not blink

Colors are associated with the following values:

Value	Color
0x0	Black
0x1	Blue
0x2	Green
0x3	Cyan
0x4	Red
0x5	Magenta
0x6	Brown
0x7	Light Grey
0x8	Dark Grey
0x9	Light Blue
0xA	Light Green
0xB	Light Cyan
0xC	Light Red
0xD	Pink
0xE	Yellow
0xF	White

So, for example, to display the ASCII Character `A` (character code `0x41`) on the screen with a black background and light grey foreground, you can manually set the memory as such:

```
*(unsigned short *)(0xB8000) = (unsigned short)0x0741;
```

In the above code, the memory adddress `0xB8000` is dereferenced as an unsigned short (16-bits) and then assigned the following value: `0x0741`.

The two bytes in this value are associted with the VGA attributes (`0x07`) and the ASCII character `A` (`0x41`)

If you execute the code above, you'll notice that the letter `A` is printed at the top-left corner of the screen. If you wanted to print the letter `B` at the top right corner of the screen, you would need consider the width of the display (80 characters) and index into memory to set the value:

```
*(unsigned short *)(0xB801E) = (unsigned short)0x1F42;
```

Since writing to direct addresses can be tedious, we can look at a few alternatives to make things easier:

```
// Create a variable to store the base address and row/column
unsigned short *vga_base = (unsigned short*)(0xB8000);
int row; // Valid values should be 0-24
int col; // valid values should be 0-79

// Direct memory assignment (base address + offset)
row = 24;
col = 0;
*(unsigned short *)(vga_base + row * 80 + col) = (unsigned short)0x7443;

// Array indexing
row = 24;
col = 79;
vga_base[row * 80 + col] = (unsigned short)0x0244;
```

Explore writing different characters with different attributes to the screen using the above approaches.

Once you have an understanding of outputting to the VGA display, we will create a set of functions to better facilitate VGA output throughout the operating system.

`vga_putc()` will be the first function that will help ensure that we can properly display text and attributes on the screen. This function accepts the following parameters: - row: row position - col: column position - bg: background color - fg: foreground color - c: character to display

When considering these different parameters, we will need to use bit shifting to set the appropriate bits for the background color and foreground color.

Our header file defines some useful macros to facilitate this:

```
#define VGA_BASE                ((unsigned short *)(0xB8000))
#define VGA_ATTR(bg, fg)        (((bg) << 4) | (fg))
#define VGA_CHAR(bg, fg, c)     (((VGA_ATTR((bg), (fg)) << 8)) | (c))
```

As characters (and eventually strings) are output to the screen, the state of the VGA display should be maintained. While the contents of the memory will remain, the driver should keep track of the current row and column, background and foreground colors, and any other detail needed.

Several functions should be implemented that modify or obtain the state so different functionality can be implemented:

- `vga_set_rowcol`
- `vga_get_row`
- `vga_get_col`
- `vga_set_bg`
- `vga_set_fg`
- `vga_get_bg`
- `vga_get_fg`

Three functions should be used to reset / clear the state of the screen: - `vga_clear`: clears all content (text) from the screen and sets the background and foreground colors - `vga_clear_bg`: clears the background color and sets it to the specified background color - `vga_clear_fg`: clears the foreground color and sets it to the specified foreground color

While all of the above are helpful for displaying individual characters and maintaining some state, there is much more value in being able to display string output, handling special characters, and subsequently incorporating support for scrolling lines so as text is printed/displayed, the display is updated naturally.

Consider that a string is simply an array of characters with a null terminator. With this in mind, we can abstract that printing a string is nothing more than iterating and printing individual characters until a null terminator is encountered.

The `vga_puts()` function should in effect print out characters one by one. To do so, it may perform calls to the `vga_putc()` function that will actually print each character.

When printing each character at the current position, the x position should increment throughout the line. When the end of the line is reached, the y position should increment and the x position should be reset back to 0.

Some special characters have certain meanings that should be considered when printing:

- `\b` (`0x08`, Backspace) should decrement the x position. If the x position is at the beginning of the line (0) then the y position should decrement and the x position set to the end of the line. If the y position is 0, then it should not decrement
- `\t` (`0x09`, Tab) should move the x position forward four (4) positions and fill each position with a space (`0x20`) character
- `\r` (`0x0D`, Carriage Return) should move the x position to 0
- `\n` (`0x0A`, New Line/Line Feed) should move increment the y position and set the x position to 0

As characters are printed on the screen and the x and y positions are incremented, eventually you will reach the last row and last column. When this happens, we want to support scrolling of text. Since all text displayed on the screen is simply data in memory, we can accomplish this by copying entire lines to other lines and also clearing out any content/text on the last line.

To visually represent on the screen where the current row/column value is located, the VGA cursor can be used. This cursor is controlled by the VGA hardware but can be configured by the driver:

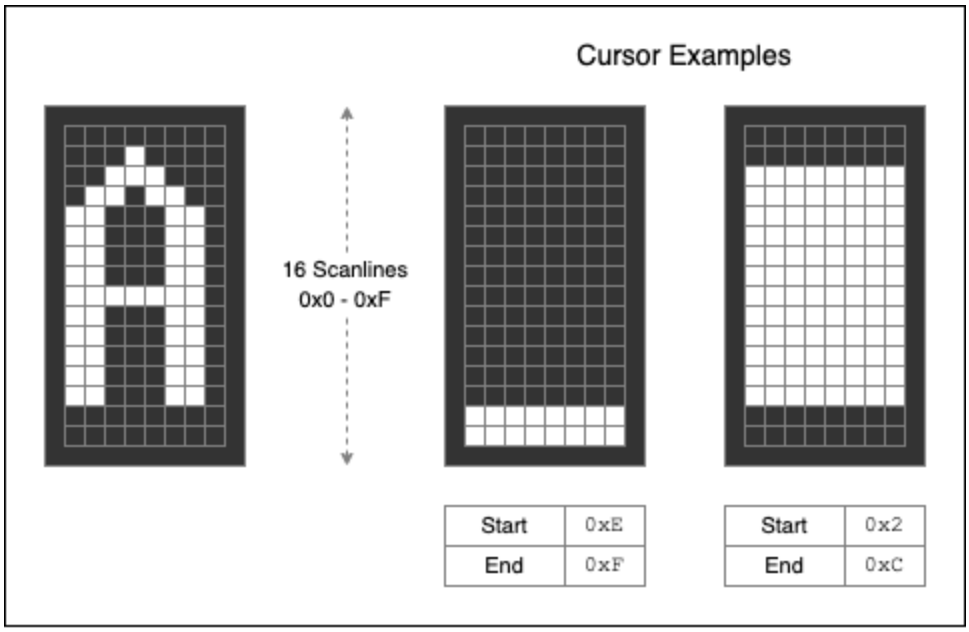
1. Enabling or disabling the cursor
2. Repositioning the cursor location
3. Configuring the cursor shape

The cursor is configured by writing to registers within the VGA controller hardware. To read or write these registers, direct input/output is performed using the `inportb` (for reading) and `outportb` (for writing) functions implemented in `io.h`.

The following functions should be implemented, as specified in the source provided:

- `vga_cursor_enabled`
- `vga_cursor_enable`
- `vga_cursor_disable`
- `vga_cursor_update`

Each of the registers are described. As you may want to customize the shape of your cursor, you can refer to the following diagram to understand the scanline definitions for the start and end scanlines to be configured in the “Cursor Start” and “Cursor End” registers.



The state of the cursor should also be maintained by the driver itself (i.e. if it is enabled or disabled).

As the row/column values are modified, if the cursor is enabled, the cursor position should be updated in hardware.

Lastly, since our VGA implementation uses several variables to maintain state, ensure that you are fully aware of what values they are defaulted/initialized to. It is recommended that you set default/known values in your `vga_init()` function which should be called from `main()` at startup.

Complete each of the following functions following the descriptions from the function headers to ensure that your VGA driver is functional.

- [ ] `vga_init`
- [ ] `vga_clear`
- [ ] `vga_clear_bg`
- [ ] `vga_clear_fg`
- [ ] `vga_cursor_disable`
- [ ] `vga_cursor_enable`
- [ ] `vga_cursor_enabled`
- [ ] `vga_get_bg`
- [ ] `vga_get_fg`
- [ ] `vga_get_x`
- [ ] `vga_get_y`
- [ ] `vga_putc`
- [ ] `vga_putc_at`
- [ ] `vga_puts`
- [ ] `vga_puts_at`

- [ ] `vga_set_bg`
- [ ] `vga_setc`
- [ ] `vga_set_fg`
- [ ] `vga_set_xy`

Optionally, you may want to make sure of the VGA text mode cursor. Several functions have been created for you already that you can use. If you choose to enable the cursor, you should ensure that as the x/y positions are updated that the cursor is moved appropriately.

Part 5: Keyboard Input

In addition to text output, we will want to support text input from the keyboard.

Keyboard input is handled by reading individual bytes with direct I/O. These bytes refer to the keyboard status or keyboard data. This status or data can be read using the following ports:

Port	Description
0x60	Keyboard Data
0x64	Keyboard Status

Reading a byte from a port can be performed using the `inportb()` function:

```
char status = inportb(0x64);
```

Reading in the status can tell us if data is available as well as other parameters of the keyboard itself. For now (especially with our emulated environment) we will simply be concerned with bit 0 which if set indicates that keyboard data is available if the bit is set.

Reading keyboard data will provide us with something called the keyboard scan code:

```
char c = inportb(0x60);
```

The keyboard scan code tells us a number of details:

1. If the key is being pressed
2. If the key is being released
3. Which specific key is being pressed or released

Bit 7, if set (`0x80`), indicates that the key is pressed. If unset, it indicates that the key is released. The remaining bits (0-6, effectively values `0x01` through `0x7f`) indicate which key is being pressed. While we are generally used to working with ASCII character codes for representing text, keep in mind that there are many other keys that do not have any direct ASCII character code associated. Additionally, keyboard data/characters also have a historical relationship to the physical hardware layout for specific keyboards. A scancode of `0x00` usually indicates an error or lack of input when reading the keyboard data port.

As a result, all keyboard scan codes need to be handled and potentially mapped to characters that would be of use for human input and output.

Any key that cannot be decoded or determined to be valid when scanning should be referred to via the defined `KEY_NULL` value.

Please review the list of Keyboard Scan Codes (Found from Canvas)

When decoding, you may want to refer to the list of ASCII Character Codes (Found from Canvas)

If we review the scan codes, we will see that there is no distinction in case for letters (example: `0x1E` is mapped to the letter `A`). Additionally, several scan codes have multiple characters associated (example: `0x0D` maps to both the = and + character).

Since keyboard scancodes come in for single key events (press or release), we need to make a distinction on how we interpret when a key is entered. Keyboard hardware will naturally support "repeat" functionality so if you hold a key down, it will appear as a sequence of press/releases.

In our implementation, we will consider that a key is entered when it is released.

While alphanumeric characters and symbols are certainly important, we should consider how other keys are handled as well. Several keys contain special meaning, such as the "CAPS LOCK" key or "SHIFT" key. This is crucial when consider that pressing the "SHIFT" key allows us to distingish between alternate representations.

As such, for some keys, we will want to maintain the state of whether or not they are pressed (or in some cases, such as with "CAPS LOCK" if it was "toggled"). This will be incredibly important when we need to decode scan codes into ASCII characters or other representations.

To support keyboard input in our operating system, we will implement the following functions:

1. `keyboard_scan()` to read the keyboard scan code from the data port.
2. `keyboard_decode()` Decodes a scancode into an ASCII character code or "special key" definition
3. `keyboard_poll()` to check if keyboard data is present, read they keyboard scan code, decode it, and return the value
4. `keyboard_getc()` to block (loop) until a valid keyboard character has been entered

Project Submission

Repositories

For phase 1 and all future phases, you will submit code as a team using your team's single project repository.

Branches

All source code must be submitted via GitHub using the `main` branch. You are welcome to utilize separate branches for your own development workflows but only source code on the `main` branch will be considered for grading.

Adding Files

Add all source files, which can be done using the `git add` command:

```
git add *
```

Commit your changes and include a commit message to describe the changes in the commit:

```
git commit
```

Push your changes to your repository:

```
git push
```

Tags

As git allows us to iteratively develop source code over a series of different commits, you will need to tag a specific commit to indicate the source code you are submitting to be graded.

All project phases must be submitted with a tag in the format of `phaseX.Y`, where `X` refers to the phase number (`1`, in this case) and `Y` refers to a submission number (usually `0`, but may increment if needing to submit code after a due date or for re-grading when permitted).

To tag a commit for phase 1 submission, use the following commands to add a tag and push the tag to your repository.

Tag the commit:

```
git tag phase1.0
```

Push the newly added tag(s):

```
git push origin --tags
```

Grading Metrics

Grading encompasses meeting functional requirements as well as submission requirements.

No credit will be given for the following:

- Lack of an appropriate tag for the commit
- Source code is submitted past the due date
- Source code cannot be compiled as-is with the original/included Makefile
- Source code has not been materially modified from the base source code or provide template
- Source code has been plagiarized, duplicated, or otherwise demonstrated to not be originally developed

The following deductions may be made:

- Up to 20 points for each missing requirement in the project phase
- Up to 10 points for modification of header files (unless indicated otherwise)
- Minimum 5 points for each incorrectly implemented functional requirement
- Minimum 5 points for each occurrence of programming errors

Examples of programming errors include (but are not limited to): - Invalid use of pointers (such as dereferencing a NULL pointer or potentially NULL pointer) - Invalid memory accesses (reading from or writing to memory that should not be accessed) - Specifically: Out-of-bounds indexing - Failures to initialize variables or memory to known/default values - Buffer overflows

Requirements Checklist

To help ensure that you have met each of the requirements for this phase, refer to this checklist below:

Part 1: Organization

- [ ] Carefully review and understand each provided header file
- [ ] Carefully review and understand each provided source code template

Part 2: Kernel Utilities

- [ ] Implement the `kernel_log_warn` function in `kernel.c`
- [ ] Implement the `kernel_log_error` function in `kernel.c`
- [ ] Implement the `kernel_log_debug` function in `kernel.c`
- [ ] Implement the `kernel_log_trace` function in `kernel.c`
- [ ] Implement/extend the `kernel_panic` function in `kernel.c`
- [ ] Implement the `bit_count` function in `bit.c`
- [ ] Implement the `bit_test` function in `bit.c`
- [ ] Implement the `bit_set` function in `bit.c`
- [ ] Implement the `bit_clear` function in `bit.c`
- [ ] Implement the `bit_toggle` function in `bit.c`
- [ ] Add a new "kernel command" (see `kernel_command` in `kernel.c`) to toggle the VGA text mode cursor
- [ ] Add a new "kernel command" (see `kernel_command` in `kernel.c`) to clear the VGA display
- [ ] Add a new "kernel command" (see `kernel_command` in `kernel.c`) to increase the kernel log level
- [ ] Add a new "kernel command" (see `kernel_command` in `kernel.c`) to decrease the kernel log level

Part 3: VGA Output

- [ ] Implement the `vga_init` function in `vga.c`
- [ ] Implement the `vga_clear` function in `vga.c`
- [ ] Implement the `vga_clear_bg` function in `vga.c`
- [ ] Implement the `vga_clear_fg` function in `vga.c`
- [ ] Implement the `vga_set_bg` function in `vga.c`
- [ ] Implement the `vga_get_bg` function in `vga.c`
- [ ] Implement the `vga_set_fg` function in `vga.c`
- [ ] Implement the `vga_get_fg` function in `vga.c`
- [ ] Implement the `vga_set_rowcol` function in `vga.c`
- [ ] Implement the `vga_get_row` function in `vga.c`
- [ ] Implement the `vga_get_col` function in `vga.c`
- [ ] Implement the `vga_setc` function in `vga.c`

- [ ] Implement the `vga_putc` function in `vga.c`
- [ ] Implement the `vga_puts` function in `vga.c`
- [ ] Implement the `vga_putc_at` function in `vga.c`
- [ ] Implement the `vga_puts_at` function in `vga.c`
- [ ] Implement the `vga_cursor_enable` function in `vga.c`
- [ ] Implement the `vga_cursor_disable` function in `vga.c`
- [ ] Implement the `vga_cursor_disabled` function in `vga.c`

Part 4: Keyboard Input

- [ ] Implement the `keyboard_init` function in `keyboard.c`
- [ ] Implement the `keyboard_scan` function in `keyboard.c`
- [ ] Implement the `keyboard_poll` function in `keyboard.c`
- [ ] Implement the `keyboard_decode` function in `keyboard.c`
- [ ] Implement the `keyboard_getc` function in `keyboard.c`
- [ ] Support distinction of characters when CAPS-lock is enabled/disabled
- [ ] Support distinction of characters when SHIFT is pressed
- [ ] Support distinction of characters when both CAPS-lock + SHIFT are pressed
- [ ] Keep track of the status of CTRL, ALT, and SHIFT characters being pressed

Project Phase 1 Submission

- [ ] Commit/submission tagged as `phase1.0`

Additional Resources

Reference Material

Project Phase 1

- OSDever FreeVGA Reference

General Reference

Working with Git

- Pro Git Book
- GitHub Git Cheat Sheet
- Git Cheat Sheet