ECE391: Computer Systems Engineering **Fall 2023**
Machine Problem 1 **Due: September [illegible]1 Mon, 5:59 PM**

Text-Mode Missile Command

In this machine problem, [illegible]mode version of Missile Command, the classic arcade video game, in x86 assembly as an ext[illegible]me clock (RTC) driver. This assignment should provide substantial experience with the x86 [illegible]duction to how drivers accomplish tasks inside the Linux kernel. This handout first explain[illegible]il, then explains several concepts that you will need to understand and implement.

[illegible]tire document before you begin.

A Note On This Handout: The sections entitled "Linux Device Driver Overview," "RTC Overview," "Ioctl Functions," and "Tasklets" contain background Linux knowledge which is not critical for you to complete this MP. The material described in these background sections will be covered in lecture in the next few weeks, but it may be helpful to read these sections to familiarize yourself with the context of your code in this MP.

Missile Command

In our version of Missile Command, you control a missile silo and try to protect your cities from enemy missiles. You direct your missiles by moving the crosshairs to the intended destination and pressing the spacebar to fire. The explosion generated at that location destroys enemy missiles within a two-square radius. The game ends when enemy missiles destroy all of your cities. Your score is the number of enemy missiles that you destroy.

The implementation of this game centers around a linked list containing an element for each active missile in the game. The game consists of two separate components: the kernel-space code, which manages this list, and the user-space code, which implements the rest of the game and processes user input. You will write a tasklet (see the section "Tasklets", below) to execute on each interrupt generated by the RTC (see "RTC Overview") and update the missiles in real-time. This linked list will reside inside the RTC driver in the kernel, so you will also write five ioctl's (see "Ioctl Functions") to provide the necessary interface between the kernel-space components of the game and the user-space components.

MP1 Data Structure

The main structure you will be working with is `struct missile`.

```
struct missile {
    struct missile* next;   /* pointer to next missile in linked list */
    int x, y;               /* x,y position on screen                 */
    int vx, vy;             /* x,y velocity vector                    */
    int dest_x, dest_y;     /* location at which the missile explodes */
    int exploded;           /* explosion duration counter             */
    char c;                 /* character to draw for this missile     */
}
```

This structure definition is usable only in C programs. There are constants defined for you at the top of the provided `mp1.S` that give you easy access to the fields in this struct from your assembly code. See the comments in `mp1.S` for further information on how to use them.

You must maintain a linked list of missiles with one `struct missile` for each active missile in the game. A variable `mp1_missile_list` has been declared in `mp1.S`. You should maintain this variable as a pointer to the first element in the linked list.

The `x` and `y` fields of the structure contain the current location of the missile on the screen. The text-mode video screen is 80×25, *i.e.*, there are 80 columns and 25 rows. In order to allow finer control over the missiles' velocities, each text-mode location is subdivided into 65536×65536 sub-squares. The low 16 bits of the `x` and `y` fields determine which of these sub-squares the missile is in, and the high 16 bits of x and y determine the text-mode video location to draw the missile. This organization makes simultaneously updating both the "game" and "screen" coordinates easy.

The vx and vy fields contain the missile's velocity, which you must use in your tasklet to update the x and y fields.

The dest_x and dest_y fields contain the missile's destination, which is the screen location at which it must explode. When the missile reaches ... op moving and begin exploding, as explained below.

The exploded field spec... the missile. When exploded == 0, the missile has not exploded and should continue mov... ither because it reached its destination or because another missile exploded nearby), this fie... number by the provided function missile_explode (see "MP1 Tasklet" for details). Whi... , your tasklet treats the missile differently. The missile should not move and should be draw... aracter defined in mp1.S. Your tasklet should also decrement this field; when it reaches zer... d your tasklet should remove the missile from the list and free the associated struct miss...

The c field specifies the character with which the missile should be drawn while it is in flight. This ability allows players to visually distinguish between enemy missiles and their own missiles.

MP1 Tasklet

The first function you need to write is called mp1_rtc_tasklet. The tasklet must update the state of the game and notify the user-level portion of the code if any cities have been destroyed or the game score has changed. Its C prototype is:

```
void mp1_rtc_tasklet (unsigned long arg);
```

Every time an RTC interrupt is generated, mp1_rtc_tasklet will be called. Your tasklet must perform three different operations. We recommend that you implement each of them as a separate function, and call those functions from mp1_rtc_tasklet.

First, your tasklet should walk down the struct missile linked list. For each missile, you should check whether it is currently exploding. If not, then you should update the x and y fields as explained above in the "MP1 Data Structure" section. There are then three cases based on the state and position of the missile.

Processing a missile requires three steps. First, if the missile has moved off of the screen (that is, its screen x coordinate is outside of the range 0-79 or its y coordinate is out of the range 0-24), then the missile should be erased from the screen, removed from the linked list, and its struct missile freed with mp1_free (see "Allocating and Freeing Memory"). Removing a missile from the list should be implemented as a separate function since you may need to perform this operation in more than one place in the code (possibly outside of the tasklet). In this document, we will refer to this function as mp1_missile_remove, though you may name it whatever you chose.

Second, if the missile has reached its destination or is currently exploding, you must call missile_explode with a pointer to the missile's struct missile as an argument. The missile_explode function (provided to you) checks whether this missile's exposion causes any other missiles or any of your cities to explode. If so, it returns a non-zero value. Otherwise, it returns zero. After calling missile_explode, you must decrement the exploded field for this missile. If it reaches zero, then the missile is finished exploding and must be erased from the screen, removed from the list, and freed with mp1_free. Otherwise, it should be drawn to the screen with the EXPLOSION character.

Finally, if the missile is simply moving toward its destination, is not exploding, and is still on the screen, you should check whether its screen position has changed. If so, you should erase it from its old position and re-draw it in its new position.

Note that in every case the missile should be re-drawn—it could have been over-written by another missile or the crosshairs moving through the same text-video location. For information on how to draw to the screen, see the "Text-Mode Video" section.

If any call made to missile_explode by the tasklet indicated that the status of the game changed (any non-zero return value), you must notify the user-space program once by calling mp1_notify_user before the tasklet returns.

After procesing all missiles, your tasklet must proceed with its second operation: redrawing the cities to ensure that any destroyed cities are drawn as destroyed. Also, if any missile has moved into a text-video location occupied by a city, the city should still be visible. The three cities should be drawn in the bottom row of the screen centered in columns 20, 40, and 60. There are two five-character arrays declared in mp1.S for you, base_pic and dead_base_pic for drawing live and destroyed cities. So, for example, the first city should be drawn in the five video locations from (18,24) to (22, 24). The base_alive array indicates whether each city has been destroyed. It contains four bytes; each of the first three is ze[illegible]ase is dead and non-zero if it is alive. The fourth byte is padding.

The third thing your taskl[illegible] crosshairs. It may have been overwritten by a missile or by a city, and we want to ensure tha[illegible]

MP1 Ioctls

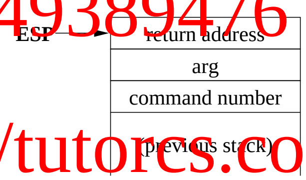
The next function you mu[illegible]ctl. Its C prototype is:

```
int mp1_ioctl (unsigned long arg, unsigned long cmd);
```

This function serves as a "dispatcher" function. It uses the cmd argument to determine which of the next five functions to jump to. The table below gives a brief summary of cmd values, the corresponding core function, and a brief description of what that core function does. Each of the core functions are described in the section entitled "Core Functions." Note that you must check this cmd value; if it is an invalid command, return -1.

cmd value	Core function	Description
0	mp1_ioctl_startgame	starts the missile command game
1	mp1_ioctl_addmissile	add a new missile
2	mp1_ioctl_movexhairs	move the crosshairs
3	mp1_ioctl_getstatus	get the current game status
4	mp1_ioctl_endgame	end the game
other	–	Any value other than 0-4 is an error. Return -1.

The method used to jump to one of the core functions is to use **assembly linkage** without modifying the stack. A picture of the stack at the beginning of mp1_ioctl is shown below.



Each of the core functions takes arg directly as its parameter. Since this parameter is passed to the mp1_ioctl function as its first parameter, mp1_ioctl can simply jump directly to the starting point of one of the core functions without modifying the stack. The arg parameter will already be the first parameter on the stack, ready to be used by the core function. In this way, it will appear to the core functions as if they were called directly from the RTC driver using the standard C calling convention without the use of this assembly linkage. Your mp1_ioctl must use a jump table—see the section "Jump Tables" below.

Core Functions

You must implement each of the following five functions in x86 assembly in the mp1.S file.

`int mp1_ioctl_startgame (unsigned long ignore);`
This function is called when the game is about to start. The parameter passed in `arg` is meaningless and should be ignored. This functio[illegible] the variables used by the driver—all of the variables declared in `mp1.S`—and the crosshai[illegible]dle of the screen: (40, 12).

`int mp1_ioctl_addmis[illegible]e* user_missile);`
This ioctl must add a new [illegible] parameter is a pointer to a `struct missile` in user space. This function needs to copy th[illegible]namically allocated buffer in kernel space. If either the dynamic memory allocation (see "[illegible]Memory" below) or the data copy (see "Moving data to/from the kernel") fails, this functio[illegible]es fail, it should be sure to free any memory it has allocated before returning. If it succeeds, i[illegible]sile into the linked list and return 0.

`int mp1_ioctl_movexhairs (unsigned long xhair_delta_packed);`
This function moves the crosshairs. The parameter is a 32-bit integer containing two signed 16-bit integers packed into its low and high words. The low 16 bits contain the amount by which the `x` component of the crosshair position should change, and the high 16 bits contain the amount by which the `y` component should change. This function should not allow the crosshairs to be moved off of the screen—that is, it should ensure that the `x` component of its position stays within the range 0-79, and its `y` component stays within the range 0-24. If the position of the crosshairs does change, this function should redraw it at its new location. It should never fail, and always return 0.

`int mp1_ioctl_getstatus (unsigned long* user_status);`
This function allows the user to retrieve the current score and the status of the three cities. The argument is a pointer to a 32-bit integer in user space. This function should copy the current score into the low 16-bits of that integer, and the status of the three cities into bits 16, 17, and 18. The `missile_explode` function maintains the user's current score in the `mp1_score` variable declared in `mp1.S`. If a city is currently alive, the corresponding bit should be a 1; if it has been destroyed, the bit should be 0. The `missile_explode` function maintains this information in the `base_alive` array, as described above. This function should return 0 if the copy to user space succeeds, and -1 if it fails.

`int mp1_ioctl_endgame (unsigned long ignore);`
Called when the game is over, this function must perform all the cleanup work. It should free all the memory being used by the linked list and then return success. When freeing the list, be careful to avoid accessing any memory that has already been freed.

Synchronization Constraints

The code (both user-level and kernel) for MP1 allows the tasklet to execute in the middle of any of the `ioctl`s, so you must be careful to order the updates properly in some of the operations. Since the tasklet does not modify the list, the main constraint is that any `ioctl` that modifies the list does so in a way that never leaves the list in an unusable state.

In particular, `mp1_ioctl_addmissile` must fill in the newly allocated structure, including the `next` field, *before* changing the head of the list to point to the new structure.

Similarly, `mp1_missile_remove` must remove the element from the list before freeing it; copying the structure's `next` pointer into a register is not sufficient, since the tasklet could try to read the structure after the call to `mp1_free`.

Getting Started

Be sure that your development environment is set up from MP0. In particular, have the base Linux kernel compiled and running on your test machine. Begin MP1 by following these steps:

- We have created a [illegible] use for this project. The repository is available at https://gitlab[illegible]ece391_fa23/mp1_<YOUR_NETID> and can be accessed [illegible]
- Access to your Git [illegible] sioned shortly after the MP is released. Watch your @illinois.edu email for an invitati[illegible]
- To use Git on a lab [illegible] use Git Bash on Windows, not the VM. You are free to download other Git tools as y[illegible] ntation assumes you are using Git Bash. To launch Git Bash, click the Start but[illegible] git bash, then click on the search result that says Git Bash.
- Run the following commands to make sure the line endings are set to LF (Unix style):
  git config --global core.autocrlf input
  git config --global core.eol lf
- Switch the path in git-bash into your z: drive by running the command: cd /z
- If you do NOT have a ssh-key configured, clone your git repo in z: drive by running the command (it will prompt you for your NETID and AD password):
  git clone https://gitlab.engr.illinois.edu/ece391_fa23/mp1_<YOUR_NETID>.git mp1
  If you do have a ssh-key configured, clone your git repo in z: drive by running the command:
  git clone git@gitlab.engr.illinois.edu:ece391_fa23/mp1_<YOUR_NETID>.git mp1

In your devel machine:

- Change directory to your MP1 working directory (cd /workdir/mp1). In that directory, you should find a file called mp1.diff. Copy the file to your Linux kernel directory with
  cp mp1.diff /workdir/source/linux-2.6.22.5
- Now change directory to the Linux kernel directory (cd /workdir/source/linux-2.6.22.5). Apply the mp1.diff patch using
  cat mp1.diff | patch -p1
  The last argument contains a digit 1, not the lowercase letter L. This command prints the contents of mp1.diff to stdout, then pipes stdout to the patch program, which applies the patch to the Linux source. You should see that the patch modified three files, drivers/char/Makefile, drivers/char/rtc.c, and include/linux/rtc.h. Do *NOT* try to re-apply the patch, even if it did not work. If it did not work, revert all 3 files to their original state using SVN (svn revert <file name>). After that, you may try to apply the patch again.
- Change directory back to /workdir/mp1. You are now ready to begin working on MP1.
- **Do not commit the Linux source or the kernel build directory. The number of files makes checking out your code take a long time. If during handin, we find the whole kernel source, any object files or the build directory in your repository, you will lose points.** We have added a .gitignore file to your initial repository. This file contains all the Git ignore rules that tells Git to not commit the specified file types. The Linux source and kernel build directory are one such example of files that are ignored. Try and explore the .gitignore file to see what other file types are ignored.

Be sure to use your repository as you work on this MP. You can use it to copy your code from your development machine to the test machine, but it's also a good idea to commit occasionally so that you protect yourself from accidental loss. Preventable losses due to unfortunate events, including disk loss, will not be met with sympathy.

Testing

Due to the critical nature of writing kernel code, it is better to test and debug as much as possible outside the kernel. For example, let's say that a new piece of code has a bug in it where it fails to check the validity of a pointer passed in to it before using it. Now, say a NULL pointer is passed in and the code attempts to dereference this NULL pointer. When running in user spac[illegible]mpt to dereference an invalid memory location and sends a signal,[1] SEGV, to the program. T[illegible]es harmlessly with a "Segmentation fault" error. However, if this same code were run insid[illegible]ould crash, and the only recourse would be to restart the machine.

In addition, debugging ke[illegible]up you developed in MP0—two machines, connected via a virtual TCP connection, with one[illegible]nd the other running a debugger. In user space, all that's necessary is a debugger. The develo[illegible]le-test-debug) in user space is much faster.

For these reasons, we hav[illegible]test harness for you to test your implementation of the additional `ioctl`s and tasklet. This [illegible]runs your code as a user-level program, allowing for a much faster development cycle, as well as protecting your test machine from crashing. Using the user-level test harness, you can iron out most of the bugs in your code from user space before integrating them into the kernel's RTC driver. The functionality is nearly identical to the functionality available if your code were running inside the kernel.

The current harness tests some of the functionality for all the ioctls, but it is not an exhaustive test. It is up to you to ensure that all the functionality works as specified, as your code will be graded with a complete set of tests.

Note: For this assignment, a test harness is provided to you that can test some of the functionality of your code prior to integration with the actual Linux kernel. Future assignments will place progressively more responsibility on you, the student, for developing test methods. What this means is that a complete test harness will not be provided for every MP, and it will be up to you to design and implement effective testing methods for your code. We encourage you to look over how the user-level test harness works for this MP, as its design may be of use to you in future MPs. This test harness is fully functional, and uses some advanced programming techniques to achieve a complete simulation of how your code will execute inside the Linux kernel. You need not understand all of these techniques; however, understanding the important ideas is useful. Questions on Piazza as to how this test harness works are welcome as well.

Running the user-level test program: To run the user-level test program, follow these steps:

- Type `cd /workdir/mp1` to change to your MP1 working directory.
- Type `make` to compile your code and the test harness.
- Type `su -c ./utest` to execute the user-level test program as `root` (you will need to type `root`'s password).

You can also type `su -c "gdb utest"` to run gdb on the user-level test harness to debug your code. Debugging the kernel code will be difficult. Use the `disas` (disassemble) command on `mp1_rtc_tasklet` or `mp1_ioctl` to see the start of your code (feel free to add more globally visible symbols), then use explicit addresses to see the rest of it. Be sure to start any disassembly with the starting byte of an instruction rather than an address in the middle of an instruction. With non-function symbols (such as those in your assembly code), and with addresses, you need an asterisk when identifying a breakpoint. For example, `break *mp1_ioctl` or `break *0x12345678`.

The test code changes the display location to the start of video memory. If you do not see a prompt after the code finishes (correctly or otherwise), pressing the Enter key will usually return the display to normal. Note also that `gdb` will return the display to its usual location, after which you will not be able to see any of the animation (while debugging).

Note: When running the user test under `gdb`, the debugger stops your program whenever a signal (such as SIGUSR1 or SIGALRM) occurs. To turn off this behavior and make it easier to debug your program, type the following commands in `gdb`:

```
handle SIGUSR1 noprint
handle SIGALRM noprint
```

[1] Think of a signal as a user-level (unprivileged) interrupt for now.

Testing your code in the kernel: Once you are confident that your code is working, you need to build it in the kernel.

- If you logged in as root to test, log out and back in again as user. If you have not already done so, commit your changes to the MP1[illegible]
- Type `cp /workd`[illegible]`ir/source/linux-2.6.22.5/drivers/char` to copy your `mp1.S` file to your l[illegible]
- Type `cd ~/build`[illegible]uild directory.
- Type `make` to buil[illegible]nges. If you have applied the `mp1.diff` file as described in the “Getting Started” s[illegible] kernel will build and link properly.
- Follow the procedu[illegible]paring Your Environment,” to install your new kernel onto the test virtual machine an[illegible]ou execute the test kernel under `gdb` when debugging.
- In the test machine[illegible]rectory using the command `cd /workdir/mp1`, then type `make clean` and `make`.
- Type `su -c ./ktest` to execute the kernel test program as `root` (you will need to type `root`'s password).

Both test programs should produce the exact same behavior.

Moving Data to/from the Kernel

Virtual memory allows each user-level program to have the illusion of its own memory address space, separate from any other user-level program and also separate from the kernel. This affords each program a level of protection, such that user-level programs cannot write to memory owned by other programs, or worse, owned by the kernel. Therefore, when passing memory addresses between a user-level program and the kernel (such as in an `ioctl` system call) a translation is needed so that the kernel can correctly reference the user-level memory address being passed to it to get at the data. This translation is performed by the `mp1_copy_to_user` and `mp1_copy_from_user` functions, which are wrappers around the real Linux kernel functions `copy_to_user` and `copy_from_user` defined in `asm-i386/uaccess.h`.

The declarations for these two functions are:

```
unsigned long mp1_copy_to_user (void *to, const void *from, unsigned long n);
unsigned long mp1_copy_from_user (void *to, const void *from, unsigned long n);
```


The semantics of `mp1_copy_to_user` and `mp1_copy_from_user` are similar to those of `memcpy`, for those of you familiar with it. These functions take two pointers to memory areas, or **buffers**, `to` and `from`, and a length `n`. Each function copies n bytes from the `from` buffer to the `to` buffer. As can be inferred from their names, `mp1_copy_to_user` copies data from a kernel buffer to a user-level buffer, and `mp1_copy_from_user` copies data from a user-level buffer to the kernel. All user- to kernel- address translations are taken care of by these functions. Each of these functions returns the number of bytes that could not be copied, which should be 0. Bad user-level pointers can cause return values greater than zero. For example, if you pass a NULL pointer in as the user-level parameter to one of these functions (such as the `to` parameter in `mp1_copy_to_user`), it checks the pointer and memory area, sees that it points to an invalid buffer, and returns `n`, since it could not copy any data.

You'll need these functions in any of the core functions which take pointers to user-level structures. Each `ioctl` takes an “arg” parameter, so you will need to look at the documentation for each `ioctl` to figure out which ones are actually pointers to user-level structures.

One final important note: When copying data to a buffer in the kernel, you should not use statically-allocated global buffers. In multiprocessor systems, for example, multiple calls to your `ioctl` functions may be going on at the same time. Using a statically-allocated storage area, like a global variable, is a bad idea because the separate calls to the `ioctl` would be contending for using this same storage area. You should use either local variables on the stack or dynamically-allocated memory. Refer to the Course Notes for information on allocating local variables on the stack. The section below has information on dynamic memory allocation in the Linux kernel.

Allocating and Freeing Memory

User-level C programs make use of the `malloc()` and `free()` C library functions to allocate memory needed for storing dynamic structures such as linked list elements. Linux kernel code uses a number of different memory allocation functions that you will learn later in the semester. Since your code must run in the kernel, you must use the memory allocation servic[illegible] provided there. To abstract the details away (for now), the MP1 distribution contains two memory allocation functi[illegible] to `malloc()` and `free()`. Their prototypes are:

```
void* mp1_malloc(un[illegible]
void mp1_free(void* [illegible]
```

`mp1_malloc` takes a para[illegible]ber of bytes of memory to allocate. It returns a `void*`, called a "void pointer," which is th[illegible] newly-allocated memory.

`mp1_free` takes a pointer [illegible] was allocated with `mp1_malloc()` and releases that memory back to the system. It does not return anything.

Text-Mode Video

Each character on the text display comprises two bytes in memory. The low byte contains the ASCII value for the character to be display. The high byte is an attribute byte, which holds information about the color of that particular character on the screen.

The screen is divided into rows and columns, with the upper-left character position referred to as row 0, column 0. Each row is 80 characters wide, and there are 25 rows. The screen is stored linearly in video memory, with each successive row stored directly after the one above it. For example, row 1, column 0 immediately follows row 0, column 79 in memory, row 2, column 0 immediately follows row 1, column 79, and so forth. Thus, to write a pixel at row 12, column 15 on the screen, you first need to calculate the row offset: row 12 $\times$ 80 characters per row $\times$ 2 bytes per character = 1920. Then, add the column offset: column 15 $\times$ 2 bytes per character = 30. So, row 12 column 15 on the screen is 1920 + 30 = 1950 bytes from the start of video memory.

mp1_poke: Due to Linux's virtualization of the screen buffer and of video memory, the start of video memory is not a constant, so writing to video memory is somewhat more complicated than using a `mov` instruction. To simplify things for this MP, a function has been defined called `mp1_poke`. This function, defined in assembly in `mp1.S`, takes care of finding the starting address of video memory and writing a single byte to an offset from that starting address. `mp1_poke` does not make use of the C calling convention discussed in the Course Notes. Instead, to use `mp1_poke`, make a function call with the following parameters:

`%eax`	offset from the start of video memory
`%cl`	ASCII code of character to write

`mp1_poke` then finds the correct starting address in memory and writes the character in CL to the location specified by EAX.

Note: For `mp1_poke`, EDX is a caller-saved register (in other words, `mp1_poke` clobbers EDX). If you need to preserve the value of EDX across a call to `mp1_poke`, you must save its value on the stack. This preservation can be accomplished by pushing the register's value onto the stack with `pushl %edx` before making the call, and then popping the value back into EDX with `popl %edx`. All other registers are callee-saved (that is, `mp1_poke` preserves their values).

Jump Tables

You must use a **jump table** to perform the “dispatching” operation in `mp1_ioctl`. A jump table is a table in memory containing the addresses of functions (called function pointers). Each function pointer is a 32-bit memory address, just like any other pointer. The memory addresses that you want to put in the jump table are the labels of the start of each function. Let’s say you have three functions in an assembly (`.S`) file, with labels `function0`, `function1`, and `function2` as the names [illegible] a jump table as follows:

```
function0:
# function 0 body

function1:
# function 1 body

function2:
# function 2 body

jump_table:
.long function0, function1, function2
```

The jump table provides an easy way to access those three functions. If you view the jump table as a C-style array of pointers:

```
void* jump_table[3];
```

`jump_table[0]` (in other words, the memory location at `jump_table` + 0 bytes) holds the address of `function0`, `jump_table[1]` (at `jump_table` + 4 bytes) holds the address of `function1`, and `jump_table[2]` (at `jump_table` + 8 bytes) holds the address of `function2`. In these examples, the number inside the brackets is the “index” into the jump table.

In this MP, the `cmd` parameter should serve as the index into the jump table, and you should be able to easily jump to each of the five core functions by creating a table similar to that shown above.

Linux Device Driver Overview

The first important concept in Linux device drivers is the fact that Linux makes all devices look like regular disk files. If you list the files in the `/dev` directory (using `ls`), you can see some devices that may be present on the machine. Each device is associated with one of the files. For example, the first serial port is associated with the device file `/dev/ttyS0`. For this MP, you will be dealing with the `/dev/rtc` device file, which is the device file associated with the real-time clock.

Since everything looks like a file, Linux drivers must support a certain set of standard file operations on their associated device files. These operations should seem familiar, as they are the operations available for normal disk files: `open`, `close`, `read`, `write`, `lseek` (to move to arbitrary locations within the file), and `poll` (to determine if data is available for reading or writing). In addition, most device files support the `ioctl` operation. `ioctl` is short for “I/O control,” and this operation is used to perform miscellaneous control and status actions that do not easily fall into one of the more standard file operations—things that you wouldn’t do to normal disk files. A good example of an `ioctl` is setting the frequency or rate of interrupts generated by the real-time clock. `ioctl`s are discussed in more depth later in this handout. It is also important to note that drivers need not support all these operations; they may choose to support only those necessary to make the device useful.

RTC Overview

A computer's real-time clock can generate interrupts at a settable frequency. Real programs running on Linux can make use of this device to perform timing-critical functionality. For example, a Tetris-style video game may need to update the positions of the falling blocks every 500 milliseconds (ms). Using the RTC, the game might set up the RTC to generate interrupts every 500 ms. Using the standard file operations above, the game can then know exactly when 500 ms has elapsed, and u[illegible]cordingly.

We now use the RTC dr[illegible] standard file operations given above map to a real device. The RTC driver uses the `ope`[illegible]s initialization and cleanup mechanisms for certain internal data structures and setup routi[illegible]bytes of data become available to be read from `/dev/rtc` on every RTC interrupt. Programs[illegible]`ll` file operations to wait for these four bytes of data to become available, thus effectively[illegible]C interrupt to be generated. The `ioctl` operation handles many other functions: setting th[illegible]TC interrupts on and off, setting alarms, and so forth.

The important concept to glean from this discussion is that drivers provide a uniform file-like interface to the outside world via their device file and the standard set of file operations described above. The internals of actually managing the device itself are left to the driver, and are not visible to normal programs. For example, in the RTC driver, no program is able to directly gain control of the RTC, manage its interrupts, and so forth. Changing the frequency is accomplished via an `ioctl`, and determining when an interrupt has been generated is done by waiting for the four bytes of data to become available to be read using `read` or `poll`.

Ioctl Functions

An `ioctl` call from a user-mode program looks like the following:

```
ioctl(int file_descriptor, int IOCTL_COMMAND, unsigned long data_argument);
```

The `file_descriptor` parameter is returned from a call to `open` on a particular file, in this case `/dev/rtc`. It is simply a number used by a program to reference a particular file that the program has opened. The program then passes this file descriptor to other functions like `ioctl`, indicating that it is `/dev/rtc` that the program wishes to operate upon.

The `IOCTL_COMMAND` parameter is the particular `ioctl` operation to be performed on the device. It is shown in caps because all `ioctl` operations are defined as constants in the header file for each device driver. All that is needed for a program to do is select the proper predefined `ioctl` command and pass that command to the `ioctl` call.

Finally, the `data_argument` parameter is an arbitrary value passed to the `ioctl`. It can be a numeric value or a pointer to a more complex structure used by the `ioctl`. The MP1 testing code passes pointers to special structures that contain all the data necessary for your RTC driver to perform the new `ioctl`s described below.

Tasklets

Interrupt handlers themselves should be as short as possible to allow the operating system to perform other time-critical tasks. Remember, when an interrupt occurs, control is immediately handed to the operating system so it can service the device. All other tasks are blocked while the interrupt handler is executing. A tasklet is a way for an interrupt handler to defer work until after the kernel has finished processing time-critical tasks and is about to return to a user program. Normally, the interrupt handler does urgent work with the device, and then schedules a tasklet to run later to do the heavier I/O or computation that takes much longer. The operating system can enable all interrupts while the tasklet is executing. The main reason for deferring this sort of work is to allow other interrupts to occur while this non-critical work is being done. This improves the responsiveness of the system.

In MP1, the RTC hardware interrupt handler schedules your tasklet (`mp1_rtc_tasklet`) to run. When the kernel is about to return from the interrupt, it calls your tasklet, which then can update the text-mode video screen, yet allow other interrupts to occur.

Coding Style and Design

In general, being able to write readable code is a skill that's just as important as being able to write working code. People and industry teams have their own preferences and rules when it comes to coding style. In this class, we won't nitpick over small things such as spaces, blank lines, or camel case, nor will we enforce any rigorous coding guidelines. However, we still do have a basic standard that we expect you to adhere to and will be enforced through grading. Our expectations [illegible]

- Give meaningful a[illegible]o long) names to your variables, labels, constants, functions, and files. Be consistent [illegible]ns.
- Do NOT use magic [illegible]t appears in your code without a comment or meaningful symbolic name). -1, 0, and 1 [illegible]d in obvious ways.
- Keep programs and [illegible]. Don't write spaghetti code that jumps back and forth everywhere. Create helper funct[illegible]asy to follow the flow of the program. Note that **helper functions must obey the C calling convention.**
- Use comments to explain the interfaces to all functions or subroutines, lengthy segments of code, and any non-obvious line of code. However, do NOT overdo it. Too many comments is just as bad as too little. Use comments to explain why, not what.

Handin

Handin will consist of a demonstration in the 391 Lab. During the demo, a TA will check the functionality of your MP, review your code, and ask some basic questions to test your understanding of the code.

Important Things to Note:

- Regardless of your assigned demo day, the deadline is the same for everyone!
- Once the deadline hits, your GitLab write access to the project will be revoked and you will not be able to push to your repositories.
- You are free to develop your own system of code organization, but we will **STRICTLY** use only the `master` branch for grading, **and will only make use of your** `mp1.S` **file.**