CS230/330: Project 1
Hello Yalnix

Handed out: Friday, January 17, 2003
Due: Friday, January 24, 2003, 11:59 p.m.

1 Assignment

In this assignment, you are going to start with nothing and develop the initial pieces of the Yalnix kernel. To do this, you have to write the yalnix master boot record loader as well as a kernel that initializes all of its interrupt and trap vectors, recognizes system calls, and runs an idle process. In addition to this, you are required to work out an initial high-level design for your kernel. This project does not require tons of programming, but it does require a reasonable understanding of the overall project. Make sure you have read the Yalnix kernel handout a few times before beginning this assignment.

2 Project Groups

Before doing anything else, you need to do the following:

1. Find a partner, come up with a group name (up to eight letters), and send us email. We will use this name to create a Unix group for your project. If you don't know what a Unix group is and how it pertains to file security, please refer to the information in the man pages for `chmod`, `chgrp`, and `newgrp`.
2. We will create a directory and repository for use by your group. After this directory has been created, modify your `.cshrc` file (or other login files) to set the `CVSROOT` environment variable to your group project repository. Please see the TAs or myself if you have any trouble setting this up.
3. Create a project directory "yalnix" in your group directory and check it into CVS.
4. Copy the file `/yalnix/sample/Makefile.template` to the `yalnix` project and rename it to `Makefile`.

You are now ready to begin.

3 Boot to the Head

Your first task is to get the Yalnix kernel off the ground and loaded into memory. To do this, you need to write a Master Boot Record (MBR) loader. When the hardware starts, it loads the MBR from the yalnix disk device and starts executing the first instruction found at `PMEM_BASE`. From here, it is your responsibility to make the hardware load the rest of the kernel. This part of the project is especially tricky, so pay careful attention to the following instructions:

1. In your yalnix project directory, create the yalnix disk file by typing the following:

   ```
   unix % ydisk format
   *** ydisk: Reformatting user file system
   Done.
   unix %
   ```

   ydisk is a special utility program found in /yalnix/bin. After the completion of this command, you should see a 16 Mbyte file in your directory called yalnixdisk. This file is the data stored by the yalnix disk device described in the main Yalnix handout.

2. Try running the hardware. To start the Yalnix simulation environment, simply type yboot. When you do this, you should see a very scary looking message like this:

   ```
   unix % yboot
   HW: Initializing memory... 4194304 bytes
   BootROM: Loading master boot record.
   BootROM: Starting execution at 1000000.
   Illegal instruction (core dumped)
   unix %
   ```

   The illegal instruction is caused because the hardware tried to execute the MBR–which you haven't written yet (it's just filled with garbage). Congratulations, your kernel has just produced its first "blue-screen of death." Let's hope that it gets better.

3. Create a file boot.c that will contain code for the boot record loader. In your file, put the following code:

   ```
   #include "hardware.h"

   int main(cpu_context_frame *f) {
      Halt();
   }
   ```

   Compile your MBR by typing 'make boot'. You should see some output kind of like this:

   ```
   unix % make boot
   gcc     -g -nostdinc -fno-builtin -I. -I/yalnix/include \
           -o yalnixmbr boot.o -nostdlib -static \
           -Wl,-M/yalnix/lib/LdMapFile.Boot -L/yalnix/lib -lboot \
   *** ydisk: installing master boot record.
       Boot record size : 80 bytes
       Text segment     : 80 bytes
       Master boot record successfully installed!
   unix %
   ```

   Now, try running your kernel again. You should see the following output:

```
unix % yboot
HW: Initializing memory... 4194304 bytes
BootROM: Loading master boot record.
BootROM: Starting execution at 1000000.

** Machine Halted **
0 seconds elapsed.
```

Congratulations. You have now created the world's most useless, but crash-free operating system.

4. Write a function `kprints(const char *s)` that prints a string to the console device so that your boot loader can actually produce some I/O. To write a single character c to the console, you will have to write some code like this (see pg. 15 of the handout):

   ```
   while (*((unsigned int *) PORT_CONSOLE));   /* Spin */
   *((unsigned int *) PORT_CONSOLE) = c;
   ```

   Note: In this case, the raw hardware is being controlled by reading and writing from the memory address `PORT_CONSOLE`. To do this in C, you have to cast the value of `PORT_CONSOLE` to a pointer and using dereferencing to read or assign a value.

   In your MBR `main()` program, add a line like this:

   ```
   kprints("Hello Boot!\n");
   ```

   Recompile your boot record loader and try running `yboot` again. You should see "Hello Boot!" printed to the screen. Congratulations, you have just controlled your first hardware device.

   Important Note: The `main()` procedure in `boot.c` has to appear as the *first* procedure in that file. Execution of the MBR begins with whatever procedure is declared first. To call `kprints()` from `main()`, you may need to use a forward declaration. For example:

   ```
   int main(cpu_context_frame *f) {
      void kprints(const char *s);

      kprints("Hello Boot!\n");
      Halt();
   }

   void kprints(const char *s) {
   ...
   }
   ```

5. Now what? Remove the `Halt()` instruction from `boot.c`, recompile and see what happens when you boot the hardware. You should get some nasty message such as

```
ABORT: Invalid exception frame: pc and sp both are 0
** Machine Halted **
0 seconds elapsed.
```

This error has occurred because the boot loader returned without setting the program counter to any kind of program—sending the CPU off a cliff to a horrible death. Another blue screen of death.

6. Add an idle procedure to your boot loader that just spins in an infinite loop.

```
void bootidle() {
   while (1);
}
```

7. Now, in `main()`, set the PC to point to the idle procedure. For example:

```
int main(cpu_context_frame *f) {
   void kprints(const char *s);
   void bootidle();

   kprints("Hello Boot!\n");
   f->pc = (void *) bootidle;
};
```

Try recompiling and running again. You will now get a different error message.

```
ABORT: Stack pointer 0x00000000 value is illegal
** Machine Halted **
0 seconds elapsed.
```

8. Set the stack pointer to some valid (hopefully unused) memory address by adding the following code to `main()`.

```
f->sp = (void *) some valid address;
```

Recompile and try running again. If you get a segmentation fault, it means that you set the SP to a bad memory address and the hardware has died. Try again.

If you get an error message like this, you are starting to make progress:

```
ABORT: Exception 1 occurred but VectorBase is NULL

** Machine Halted **
1 seconds elapsed.
```

9. The weird error message about `VectorBase` is being caused because the hardware generated an interrupt, but you didn't define a interrupt handler function. To deal with this, you need to write an interrupt handler and set up the hardware interrupt vector. To do this, write a function like this:

   ```
   void null_handler(cpu_context_frame *f) {
      kprints("INT\n");
   }
   ```

   Now, define a dummy interrupt vector array in `main()` like this:

   ```
   void main(cpu_context_frame *f) {
      void null_handler(cpu_context_frame *);   /* Forward declaration */
      static void (*ivector[TRAP_VECTOR_SIZE])(cpu_context_frame *f) = {
            null_handler, null_handler, null_handler, null_handler,
            null_handler, null_handler, null_handler, null_handler };
      ...
      *((unsigned int *) PORT_VECTOR_BASE) = (unsigned int) ivector;
   }

   void null_handler(cpu_context_frame *f) {
   ...
   }
   ```

   Recompile and run again. You should now see output like this:

   ```
   HW: Initializing memory... 4194304 bytes
   BootROM: Loading master boot record.
   BootROM: Starting execution at 1000000.
   Hello Boot!
   INT
   INT
   INT
   INT
   ...
   ```

   Congratulations, the hardware is now running and it hasn't crashed. So what is the hardware actually doing? Well, the CPU is mostly running the infinite loop you defined in `bootidle()`. The `INT` messages are being generated by a periodic clock interrupt and occur about once per second. If the Yalnix graphics window appears, try pressing the mouse button and moving the mouse around. You should now see a *LOT* of `INT` messages—these interrupts are being generated by the pen device.

4 Loading the Kernel

During boot-up, the hardware loads the master boot record into memory starting at `PMEM_BASE`. This is the first page of physical memory. Since the MBR is only one disk block, it is very small (only 1024 bytes in fact). Your next task is to modify the boot loader to bring the rest of the kernel into memory from the disk. The main yalnix kernel image is linked to start at memory address `PMEM_BASE + PAGESIZE`. Therefore, the main task of your loader is to issue commands to the disk device to get the rest of the kernel loaded starting at this address. This means that you're going to have to talk to the hardware. Pay very careful attention to these instructions.

1. Write a `kernel.c` file that defines a rudimentary kernel `main()` function. This will be almost identical to the one in your `boot.c` file. For example:

   ```
   void main(cpu_context_frame *f) {
      kprints("Hello Yalnix\n");
   }
   ```

   You will need to copy the `kprints()` function from `boot.c` to `kernel.c` (`boot.c` and `kernel.c` are two separate programs).

2. Compile your kernel by typing 'make yalnix'. If successful, you should see some output that looks similar to this:

   ```
   *** ydisk: installing kernel image.
       Kernel image size : 8196 bytes
       Text segment      : 168 bytes
       Kernel image successfully installed!
   ```

   This means that you kernel has been installed on the Yalnix disk device. Your next task is to get the boot loader to load it.

3. Returning to `boot.c`. Write a rudimentary interrupt handler for the Yalnix disk device like this:

   ```
   void trap_disk(cpu_context_frame *f) {
     kprints("DISK\n");
   }
   ```

   Now, in the boot `main()`, install the disk handler in the interrupt vector and issue a command to the disk device to make it read block 1 of the disk so that it is loaded into memory at address `PMEM_BASE + DISK_BLOCKSIZE`. Since the MBR is smaller than a disk block, this memory address is just above the MBR, but still located in the first page of memory. For example:

   ```
   void main(cpu_context_frame *f) {
      void trap_disk(cpu_context_frame *f);
      ...

      ivector[TRAP_DISK] = trap_disk;
      *((unsigned int *) PORT_DISK_ADDR) = PMEM_BASE+DISK_BLOCKSIZE;
      *((unsigned int *) PORT_DISK_CS)   = (DISK_READ << 28) | 41;
      ...
   }
   ```

Recompile and run. You should see output from your disk handler like this:

```
HW: Initializing memory... 4194304 bytes
BootROM: Loading master boot record.
BootROM: Starting execution at 1000000.
Hello Boot!
DISK
INT
INT
INT
```

Notice how a `DISK` interrupt now shows up once (to signal the completion of the disk read).

4. Oh my, could this get any worse? Now that you have read block 41 from the disk, you need to look at it to get information about the kernel. As it turns out, block 1 contains the YOFF header for the Yalnix kernel image (see pg. 20 of the Yalnix handout). Using this, you need to bring the rest of the kernel into memory by loading it from blocks 42-n where n is the last block of the kernel (which you have to compute). **Note: please refer to section 6 of the Yalnix handout for details on the disk layout.**

   To make this work, you need to make some modifications to the disk interrupt handler `trap_disk()`.

5. Add the following header file to `boot.c`.

```
#include "yoff.h"
```

6. Compute the number of blocks in the kernel image. Do this in `trap_disk()` as follows:

```
void trap_disk(cpu_context_frame *f) {
   yoff_header *h = (yoff_header *) (PMEM_BASE + DISK_BLOCKSIZE);
   int  nblocks = (h->length >> 10) + 1;
   kprints("DISK\n");
}
```

7. Define a counter from keeping track of how many blocks have been loaded as well as an address pointer. Make these variables global or static.

```
void trap_disk(cpu_context_frame *f) {
   static int   block_count = 0;
   static char *block_addr = (char *) (PMEM_BASE + PAGESIZE);

   ...
}
```

8. Now, modify `trap_disk()` to start bringing the kernel into memory. The kernel starts at block 42 and continues for `nblocks` (computed above). To load the kernel, you need to issue read commands to the disk device to bring the kernel into memory one block at a time. For example:

```
void trap_disk(cpu_context_frame *f) {
   ...
   if (block_count < nblocks) {
      *((unsigned int *) PORT_DISK_ADDR) = (unsigned int) block_addr;
      *((unsigned int *) PORT_DISK_CS) = (DISK_READ << 28) | (block_count+42);
      block_addr += DISK_BLOCKSIZE;
      block_count++;
   } else {
      kprints("DONE!\n");
   }
   ...
}
```

Make the above modifications and recompile your boot loader. When you run it, you should see some output like this:

```
HW: Initializing memory... 4194304 bytes
BootROM: Loading master boot record.
BootROM: Starting execution at 1000000.
Hello Boot!
DISK
DISK
DISK
DISK
DISK
DONE!
INT
INT
INT
...
```

Notice the sequence of DISK interrupts. If you've made it this far, you've almost got it!

9. Modify the boot loader to call kernel `main()`. In the code that prints "DONE!" above, add this extra horrible looking code:

```
...
else {
   void (*entry)(cpu_context_frame *);
   kprints("DONE!\n");
   entry = (void (*)(cpu_context_frame *)) h->text_entry;
   (*entry)(f);  /* Call the kernel */
}
...
```

This code extracts the entry point to the kernel from the YOFF header and calls it. Recompile your boot record and run it. If it works, you will now see the following:

```
HW: Initializing memory... 4194304 bytes
BootROM: Loading master boot record.
BootROM: Starting execution at 1000000.
Hello Boot!
DISK
DISK
DISK
DISK
DISK
DONE!
Hello Yalnix!
INT
INT
INT
```

Holy cow! You have just booted your yalnix kernel. That "Hello Yalnix!" message is coming from the code your wrote in kernel.c. Take a rest and meditate on the extreme bogosity you have just witnessed.

5 Off and Running!

Now that you've got the main part of your kernel to load and execute, it's time to start cranking on the real parts of your kernel. To do this, the kernel is first going to reinitialize all of the interrupts and decouple itself from the bogosity of the boot loader. To do this, follow these steps:

1. Write some functions to help you debug your kernel. In the first part of the project you wrote `kprints()` to get some output on the console. You're now going to expand this by writing a few other utility functions. First, implement a function `kputchar(int c)` that writes a single character to the console port. Now, write a collection of functions like this:

   - `kprints(char *s)`. This function prints a normal string to the console using your kputchar() function.
   - `kprintd(int d)`. Print a decimal number to the console.
   - `kprinth(unsigned int h)`. Print an unsigned number in hexadecimal to the console.

2. Modify your kernel so that it reads the amount of physical memory from the `PORT_PHYSM` port and prints out the value using your `kprintd()` function.

3. Modify your kernel so that it prints the value of the `cpu_context_frame *` parameter as a hexadecimal number using your `kprinth()` function.

4. At this point, the output of your kernel should look something like this:

```
HW: Initializing memory... 4194304 bytes
BootROM: Loading master boot record.
BootROM: Starting execution at 1000000.
```

```
Hello Boot!
DISK
DISK
DISK
DISK
DISK
DONE!
Hello Yalnix!
memory = 4194304
frame  = 0xffbdd858
INT
INT
INT
```

5. Write dummy trap (interrupt) handling functions that simply prints out a message indicating the type of interrupt and the current value of the program counter. Modify kernel `main()` to initialize the exception vector to point at these dummy routines. Note: don't forget to store the address of the exception vector into `PORT_VECTOR_BASE`. The output of each interrupt handler should appear something like this:

   ```
   TRAP_CLOCK : pc = 0x3265d8
   TRAP_MEMORY : pc = 0x3265d8
   ... etc ...
   ```

   When you overwrite the exception vector, the output of your kernel should suddenly change. For example:

   ```
   HW: Initializing memory... 4194304 bytes
   BootROM: Loading master boot record.
   BootROM: Starting execution at 1000000.
   Hello Boot!
   DISK
   DISK
   DISK
   DISK
   DISK
   DONE!
   Hello Yalnix!
   memory = 4194304
   frame  = 0xffbdd858
   TRAP_CLOCK : pc = 0x3265d8
   TRAP_CLOCK : pc = 0x3265d8
   TRAP_CLOCK : pc = 0x3265d8
   TRAP_CLOCK : pc = 0x3265d8
   ...
   ```

   This simply means that the boot loader is no longer handling interrupts–instead your real yalnix kernel has taken over.

6. Write a simple in-kernel idle process that simply spins in an infinite loop like this:

```
void KernelIdleLoop() {
     while(1) {
          Pause();
     }
}
```

7. Get yalnix to jump to this function after you exit kernel `main`. To do this, you will need to set the program counter (PC) field in the context structure (passed to the `main` routine) to point to your `main` function. In addition, you will need to set the stack pointer (SP) field to a suitable location (you will need to experiment). This is just like setting up the idle loop in the boot loader. After setting these values, simply return from `main`. Now, run your kernel. You should see the timer interrupt calling your exception handler every second. If you're feeling lucky, go to the Yalnix display window and move the mouse around with the button down. You should see a whole bunch of interrupts being generated in your kernel in addition to the clock interrupt. Congratulations, you have just created the first Yalnix process!

   Note: please make sure you use the `Pause()` instruction as shown above. This makes the class machines run a little faster.

8. Implement dummy handling functions for all of the system calls required in the project (FORK, EXEC, etc...). These handler functions should do nothing more than print the name of the system call and return. Modify the trap handler for `TRAP_KERNEL` to invoke the appropriate system call handler. You should also check for invalid/unknown system calls and print an appropriate error message. Note : the system call numbers can be found in the file `/yalnix/include/yalnixcall.h`.

9. To test all of your trap vectors and system calls, you can run your kernel by booting it using "yboot -p". This option pings all of your kernel's interrupt vectors and system calls to make sure you have covered all of the entry points to the kernel. If you have done everything correctly, your kernel will print a series of messages for each trap and system call.

10. For fun, see if you can get the pen and display devices to do anything. In your `TrapPen` interrupt handler, read the value of `PORT_PEN_EVENT` to find out where the user touched the screen. Then write a command to `PORT_DISPLAY_CTRL` to plot a single pixel on the display. Now, hold down the mouse button and move the mouse around in the Yalnix window. You should see a bunch of points being drawn.

6 Organizing your work

Once you have everything in the first part working, your next step is to complete the following steps:

1. Go through the kernel project handout and make a list of all major subsystems and subcomponents that will form the implementation of your kernel. In your list, you should list all of the system calls and traps that make up each subcomponent.

2. Next, break up your solution to the first part into a collection of files that correspond to the subsystems in the previous step. For example, instead of having everything in `kernel.c`, you should have files such as 'process.c' for process management system calls, 'serial.c' for interrupt handlers pertaining to the serial device, and so forth.

3. Once your have broken up your kernel into files, create a header file for *each* of your kernel subcomponents (e.g., `process.h`, `serial.h`, etc...). Within each header file, provide the C prototypes for all of the functions that you are going to include in the associated `.c` file. You should do this by making declarations such as the following:

   - `extern void TrapMathHandler(cpu_context_frame *);`

   As a general rule, students have found it easier to declare all trap handlers and system calls with exactly the same prototype. For example:

   ```
   extern void TrapMemory(cpu_context_frame *);
   extern void SysFork(cpu_context_frame *);
   extern void SysExec(cpu_context_frame *);
   ... etc ...
   ```

   Please note that inside the kernel, the function prototypes and names of the system calls do not have to match the functions that user programs will use to access system calls. This is because user programs do not directly link against your kernel functions. Instead they issue a trap which is handled through your `TRAP_KERNEL` handler function.

4. Modify each `.c` file so that it includes its corresponding `.h` file.

5. Modify your Makefile and any other related functions so that your kernel compiles and works exactly as it did when it was all in one big file.

6. In your README file, write a short listing of *every* file in your kernel along with a short description of the system calls, traps, and additional functionality that you plan to implement.

7 Testing

A reference kernel in the form of a yalnix disk image is available in the directory `/yalnix/kernel`. To run this kernel, simply copy the file 'refdisk' to your own directory and type `yboot -d refdisk`. You can run this kernel to view the desired behavior of what you will be implementing (note : you must run the reference kernel from your own directory). The correct output of the kernel ping test should appear something like this:

```
% yboot -p
HW: Initializing memory... 4194304 bytes
BootROM: Loading master boot record.
BootROM: Starting execution at 1000000.
Hello Boot!
DISK
DISK
DISK
```

```
DISK
DISK
DONE!
Hello Yalnix!
memory = 4194304
frame = 0xffbdd858
Initializing traps.

*** Pinging the interrupt vectors...
TRAP_CLOCK   : pc = 0x0100282c
TRAP_ILLEGAL : pc = 0x0100282c
TRAP_MEMORY  : pc = 0x0100282c
TRAP_MATH    : pc = 0x0100282c
TRAP_SERIAL  : pc = 0x0100282c
TRAP_PEN     : pc = 0x0100282c
TRAP_DISK    : pc = 0x0100282c

*** Testing system calls...
TRAP_KERNEL  : pc = 0x0100282c
SysFork, frame = 0xffbdd858
TRAP_KERNEL  : pc = 0x0100282c
SysExec, frame = 0xffbdd858
TRAP_KERNEL  : pc = 0x0100282c
SysExit, frame = 0xffbdd858
TRAP_KERNEL  : pc = 0x0100282c
SysWait, frame = 0xffbdd858
TRAP_KERNEL  : pc = 0x0100282c
SysBrk, frame =  = 0xffbdd858
TRAP_KERNEL  : pc = 0x0100282c
SysDelay, frame = 0xffbdd858
TRAP_KERNEL  : pc = 0x0100282c
SysGetPid, frame = 0xffbdd858
TRAP_KERNEL  : pc = 0x0100282c
SysYield, frame = 0xffbdd858
TRAP_KERNEL  : pc = 0x0100282c
SysSerialRead, frame = 0xffbdd858
TRAP_KERNEL  : pc = 0x0100282c
SysSerialWrite, frame = 0xffbdd858
TRAP_KERNEL  : pc = 0x0100282c
SysPenRead, frame = 0xffbdd858
TRAP_KERNEL  : pc = 0x0100282c
SysDisplayWrite, frame = 0xffbdd858
TRAP_KERNEL  : pc = 0x0100282c
SysSemaCreate, frame = 0xffbdd858
TRAP_KERNEL  : pc = 0x0100282c
SysSemaSignal, frame = 0xffbdd858
TRAP_KERNEL  : pc = 0x0100282c
SysSemaWait, frame = 0xffbdd858
TRAP_KERNEL  : pc = 0x0100282c
SysSemaDestroy, frame = 0xffbdd858
TRAP_KERNEL  : pc = 0x0100282c
SysSemaDestroy, frame = 0xffbdd858
TRAP_KERNEL  : pc = 0x0100282c
```

```
...
...
TRAP_KERNEL  : pc = 0x0100282c
SysConsoleWrite, frame =  = 0xffbdd858
TRAP_KERNEL  : pc = 0x0100282c
UNDEFINED SYSTEM CALL
TRAP_KERNEL  : pc = 0x0100282c
UNDEFINED SYSTEM CALL
0 Kernel Pages in use.

** Machine Halted **
4 seconds elapsed.
%
```

8 Handin procedure

Your project must be handed in via CVS as follows:

- Make sure all of the files needed to recreate your kernel (source code, header files, and makefiles) have been added to a CVS project named 'yalnix.' You will need to use the `cvs add` command to do this.
- Add your README file to the repository. Your README file should contain all of the names and email addresses of your team members. In addition, it should include the short writeup previously described.
- Make sure you commit all of your files to the repository using `cvs commit`.
- Test your handin by going to a temporary directory and trying it out:

```
% cvs checkout yalnix
...
% cd yalnix
% make
% yboot
```

- Tag your handin as 'project1' by typing '`cvs tag project1 yalnix`' in the directory that contains your 'yalnix' project directory (e.g., you should not type this while inside your yalnix project directory itself).
- You're done. We will check out your project by doing a '`cvs checkout -r project1 yalnix`' on your repository.

9 Implementation Hints

- The C prototype for all of your trap handlers is as follows:

```
void InterruptHandler(cpu_context_frame *);
```

- The interrupt vector should be created as an array of pointers to handler functions. The C syntax for doing this is as follows:

  ```
  void (*ivector[TRAP_VECTOR_SIZE])(cpu_context_frame *);
  ```

  Alternative, you can also do it with typedefs like this:

  ```
  typedef void (*handlerfunction)(cpu_context_frame *);
  handlerfunction ivector[TRAP_VECTOR_SIZE];
  ```

  Please refer to the K&R C programming for gory details on function pointers.

- To debug your kernel after a crash, there are a couple of things you can do. First, you can type '/yalnix/bin/ydb' to produce a simple stack trace. Alternatively, you can run gdb on the core file. To do this, you must run gdb as '`gdb yalnix core`'.

- Debugging your master boot record will be nearly impossible if you don't follow the instructions contained in this handout. Please come see us if you are having trouble with that part of the assignment.

- A careful reading of `hardware.h` and the Yalnix handout will reveal that the following code (and all similar code)

  ```
  while (*((unsigned int *) PORT_CONSOLE));
  *((unsigned int *) PORT_CONSOLE) = (unsigned int) c;
  ```

  can be replaced by the following:

  ```
  while (ReadPort(PORT_CONSOLE));
  WritePort(PORT_CONSOLE,c);
  ```

- Start early. You can probably finish this project in a couple of hours if you are really on top of everything. However, you probably don't want to wait until the last minute.