

# SFWR ENG 3F03 Summary

---

Author: Kemal Ahmed

Instructor: Dr. Ned<sup>2</sup>

Date: Winter 2014

Please join GitHub and contribute to this document. There is a guide on how to do this on my GitHub.

UNIX Commands .....	2
Regular Expressions.....	2
...commands continued .....	2
Conditional Blocks.....	6
Loop Examples .....	6
Extra Bash Stuff.....	7
Makefiles.....	7
Processor Information.....	9
Registers.....	9
Assembly.....	10
Some Commands .....	10
Stack.....	12
Segments/subsections .....	13
Directives .....	13
Data Directives.....	14
Types.....	15
Reliance on a C Driver.....	15
Compiling .....	15
Pointers compared to C.....	16
Recursion .....	16
C.....	16
Assembly.....	16
C.....	16
Variables .....	16

Dynamic .....	17
Operators .....	17
Arrays .....	17
Other stuff .....	17
Personal Questions .....	17
Shifts .....	17
Profiling .....	17

## UNIX Commands

UNIX commands are a set of commands saved on UNIX systems that let you perform many tasks that you may want to execute.

There are a number of ways you can run UNIX commands:

- **Terminal:** a program that is shell that runs commands as they are manually entered
- **Bash files** are scripts that automate the execution of a number of UNIX commands
  - Represented by .sh

## Regular Expressions

Regular expressions (regexp) are one of the most important thing to know in terms of knowing how to use UNIX commands, especially learning how to use the **Kleene star** [\*].

Remember from SE 2FA3 ([for the tron kids who never took that course](#))

## ...commands continued

When you first open your terminal, it says your username followed by “ ~”. This is because the tilde represents your username. It’s redundant to say the same name multiple times.

In this document, I will represent a field by %field%, where the word in between the two percent signs will represent the description of what should go in the field

%command% -%option(s)% %target%

you can stack options, too

e.g. list files in directory: `ls`

options:

- `r`: reverse order
- `l`: display modified date and permissions
- `t`: sort by modified date
- `a`: hidden files
- more on the man page

all options:

```
ls -ltr Documents/Dropbox
```

Notice how the target is `Documents/Dropbox`, although the full directory name would be `/user/Documents/Dropbox`

This is because the default present working directory (`pwd`) is `/user/`  
From a given current working directory, it is assumed (unless specified) that you are referring to a directory in the `pwd`.

The default lowest drive name is `/`

`%command%` `man`: gives the man page, which explains what the command does and hopefully all the options and what they do.

`$(variable)`: retrieves the value of a variable, whereas normally without the `$` it would be interpreted as a file/directory

There are some variables made by default, known as environment variables:

- `pwd`: variable that holds your present working directory
  - it is also a command that is the equivalent of `echo $pwd`
  - for some reason, `echo $pwd` doesn't work properly in terminal
- `HOME`: variable that holds the location of your home folder, i.e. where are all your personal user files saved

View all your environment variables by the command `printenv`.

To store a value in a variable, simply do: `%variable name% = %value%`

Sometimes you may have folders or files with spaces in them. Unfortunately, each word will usually be treated as a different file. To avoid this, cluster everything in quotations, e.g. `"My Documents/ENG III/SFWR ENG 3F03/a1.sh"`

`cd %target%`: change working directory to the target

`rm`: delete file

- `-i`: verify deletion
- `-v`: detail successful deletion
- `-r`: recursive delete; deletes sub-directories in the folder

`echo`: prints the value of the variable or word in the terminal

`mv`: moves file, a.k.a. "cut"

`cut %target%`: extracts parts of a line of input, usually a file

- `-c %character min #-%char max #%` `%file%`: extracts characters min to max from each line in a file
- if you don't include a max #, it will extract to the end of the line; if you don't include a min #, it will extract from the beginning of the line

- `-d '%delimiter%':` parses

`cp %source% %target%:` copies files and directories from source to target; if the target doesn't exist, it is created

options:

- `-r` or `-R`: recursively copies, i.e. includes sub-directories if it's a folder
- `-H`: doesn't actually copy the directory by having 2 versions of the same data, but rather makes a **symbolic link**, which means it just creates another pointer to the same location. Note: this is different from a shortcut because a shortcut is a program that opens the folder from a different location, whereas a symbolic link works as if it is a folder and can even be worked from
  - e.g. if I make a symbolic link of my directory, phone, and call it mobile, I can `cd` into phone OR mobile

`find`: finds a file

options:

- `-maxdepth %n%:` only look inside  $n-1$  levels of directories (i.e.  $n$ , including `pwd`)
- `-mtime %days%:` find a file modified within a certain day range. Here are the definitions of the day ranges (1 day = 24 hours, ignore Gregorian days):
  - `0`: past 1 day ( $\text{now}-1 < \text{date} < \text{now}$ )
  - `-%n%:` modified less than  $n$  days ago ( $\text{now}-n < \text{date} \leq \text{now}$ )
  - `+%n%:` modified more than  $n$  days ago ( $\text{date} < \text{now}-n$ )
  - `%n%:` modified on the day  $n$  days ago ( $\text{now}-n-1 < \text{date} < \text{now}-n$ )
  - You can also stack conditions, like if you want to find a date that is between 3-5 days ago, you could say: `-mtime +5 -mtime -3`
- `-mmin %minutes%:` same as `mtime`, but instead of day range, it is minute range
- `-name %name%:` finds a file of a given name
- `-type %type%:` find a file of a given type
  - `d`: directorys
  - `f`: file
  - `l`: symlink
  - `s`: socket

`wc`: prints the number of words, new lines, and bytes in a file

- `-l`: line count
- `-c`: byte count
- `-m`: character count
- `-L`: length of longest line
- `-w`: word count

`sed`: string processing:

- reads a line from a file into a buffer
- executes the command(s) on the buffer
- outputs the buffer to the standard output

If you push any data, you MUST pop it, since terminal does not have the garbage collection necessary to avoid that.

Remember [SFWR ENG 3GA3](#)? When calling a label (i.e. function), the address of the label will be pushed onto the stack and saved in RA (Return Address). "Return" will pop the return address from the stack.

nm: gives you the functions inside an object file  
Undefined functions are functions called, but not located within the file, such as standard functions, e.g. printf

scanf: inputs string with the formatting tags of printf

lea: load effective address

which %command%: shows the location of where the command is stored

printf: prints and formats using regular sets as well as its own set of commands; it doesn't know the number of arguments

ps: gives you a list of your running tasks as well as the PID for each (Process IDentification number)

kill -%signal #% %PID%: terminates a process

chmod: this lets you change permissions for files. There are 3 types of permissions: read, write, and execute. There are also 3 groups of users you can change the permissions to: owner, group, and other. To change the permissions for a file you need to enter the appropriate code. Use [this](#) to determine the code.

OR

Think of each of the 3 numbers in binary:

All permissions 777  $\Rightarrow$   $\underbrace{111}_{\text{owner}} \underbrace{111}_{\text{group}} \underbrace{111}_{\text{other}}$   
 $\begin{array}{ccc} 1 & 1 & 1 \\ \downarrow & \downarrow & \downarrow \\ \text{read} & \text{write} & \text{execute} \end{array}$

e.g. Only owner write: 200  $\Rightarrow$  010,000,000

If you remove execute permissions to yourself of a directory, you can create files in the directory, list contents (ls), and list permissions of the contents (ls -l). However, you cannot change into the directory (cd).

You can save the result of a command in a file:

%command% > %target file name, inc. extension%

This is especially useful when concatenating two files:

cat file1.c file2.o > file3.txt

Two accents (>>) instead of one (>) will open the file after the command is executed.

`free`: free, used, and total RAM, swap in KB

- `-m`: floors to MB

**Pipelining: ?**

## Conditional Blocks

conditional blocks are ended with `fi`:

```
if [ %condition 1% ]
%command set 1%
elif [ %condition 2% ]
%command set 2%
else
%command set 3%
fi
```

Use the `test` command:

```
if test %condition%
```

OR the equivalent is using square brackets:

```
if [ %condition% ]
```

Some comparisons include:

- OR: `%cond1% -o %cond2%`
- AND: `%cond1% -a %cond2%`
- NOT: `[ ! %cond% ]` OR `! [ %cond% ]`
- file exists: `-e %file%`
- file exists and is readable: `-r %file%`
- file exists and is writable: `-w %file%`
- checks if it is a directory (as opposed to a file): `-d %file%`
- checks if it is a file (as opposed to a directory): `-f %file%`
- is executable?: `-x %file%`
- file size > 0: `-s %file%`

```
case $%case%
    %value 1 of case% ) %command 1% ;;
    %value 2 of case% ) %command 2% ;;
esac
```

## Loop Examples

```
for i in $files
do
echo "Echoing file name: " $i
done
```

```
COUNTER=0
while [ $COUNTER -lt 10 ]; do
echo The counter is $COUNTER
let COUNTER=COUNTER+1
done
```

## Extra Bash Stuff

Start every bash script with the following line: `#!/bin/bash`

Bash debug mode should begin with: `#!/bin/sh -x`

One of the useful parts of bash files is that you can execute functions.

If users add arguments when executing the script in command line, some default variables include:

- `$0`: name of the script
- `$1`: argument 1
- `$2`: arg 2
- etc.
- `$FILES`: a set containing all arguments

You can make files that identify things that run before each shell:

- `.bashrc`: while user is logging in
- `.bashrc_profile`: while user is opening a shell

`$?`: outputs errors from previous line

## Makefiles

Makefile is a utility that compiles a set of source code (usually a project), like a shell-script extension of bash, since it uses UNIX commands. This is especially useful for large projects. Use Makefiles to automate removing libraries from directory before using SVN. It is useful for any type of source that can be compiled with bash, but the most common application is compiling C/C++ source code.

If you have multiple files in a project that refer to one another, such as objects, you usually compile them into their individual object files. However, they don't actually refer to anything other than symbols, until you use an overall compiler to join them into one main executable. Makefiles can compile and will also join the files together.

Whatever format you use to save the source for the Makefile, you'll need to compile that, using the command:

```
make -f MyMakefile
```

Similar to bash, signify different elements in a list using spaces

Declare:

- LIBS: folders
- INCLUDE\_PATH:

Simply running the make command in the shell will run the first target in the file.

Makefiles have executable sections, similar to functions, known as **targets**. Each target is saved as an executable, so when you run a target, it'll only run it if you don't have a file that already has that name. When you execute a target, you run off a directory. To execute a target from the shell, run:

```
make target
```

However, if you want a function that does not save as executable, such as "all", at the very top above the variables, put

```
.PHONY all
```

If you have multiple executables you want to compile simultaneously, place a target called "all" at the top of your file that refers to each of your targets (one for each project)

Access Makefile variables with \$(var\_name)

Access bash variables with \$\$var\_name

Compile using: `gcc -c *.c -I../`

This means: use the c-compiler from gcc on all c files (although you could just specify which file(s)) that are Included in the given directory

The compiled files have the extension .o

You'll need to move all the object files to one central location before you make your executable from them.

You should also include a section to remove your previous files, called "clean"

.inc represents a library

driver.c calls assembly files

include a target called `clean` that deletes all the files and is only activated when `make clean` is run.

declare any dependencies of .c files, like .h files by saying:

```
DEPS = stdio.h
```

If you have multiple files:

- you can save time, by setting a c compiler flag:
  - `CC=gcc`
  - and then referring it by `$(CC)`



- also, for each file (file.c) that will be compiled in your Makefile, make a separate target (called file.o)
  - you can refer to the name of the target by saying `$@`
  - you can refer to all pre-requisites by saying `$?`

By default, a makefile with just `target: fileA.o fileB.o` will run:

- `cc -c -o fileA.o fileA.c`
- `cc -c -o fileB.o fileB.c`
- `cc -o fileA.o fileB.o target`

## Processor Information

There are two main processors we're going to look at:

- 8086
- 80386+
- 80286

## Registers

8086: Real Mode

Memory  $\leq 2^{20}$  bytes = 1MB

$16 \times \text{selector} + \text{offset} \leftarrow$  optimization of finding the address

16-bit registers:

- General Purpose:
  - Other Registers: **AX, BX, CX, DX**
    - Accumulator, **B**ase, **C**ounter, **D**ata
    - all can be decomposed into an upper and lower part, e.g.  $AX = [AH|AL]$  (**H**igher and **L**ower part)
  - Index registers: **SI, DI**
    - **S**ource, **D**estination
  - Pointers to data in the machine language stack: **BP, SP**
    - **B**ase, top of **S**tack
- Segment registers:
  - **CS, DS, SS, ES**
  - **C**ode, **D**ata, **S**tack, **E**xtra
- IP: Instruction Pointer, address of next instruction to be executed
- FLAGS: a register with bits that store results of operations and processor state
  - ZF: zero flag
  - Unsigned integers
    - CF: carry flag
  - Signed integers
    - OF: overflow flag (1 if overflow)
    - SF: sign flag (1 if negative)

Each register's address is saved

**Selector:** is an index into a descriptor table

- segment register
- segments are useful if you don't have enough RAM

**Offset:**

Non-unique address

80386+:

- 32-bit registers: (Extended)
  - Operational
    - EAX, EBX, ECX, EDX
    - ESI, EDI: Source, Destination
  - Functional:
    - EBP, ESP, EIP
    - Base Pointer: base of program
    - Stack Pointer: points to where stack is stored
    - Index Pointer:
- 16-bit registers:
  - General segment registers: FS, GS

80286:

- processor arrangement that is used by Windows, Mac, and Linux
- segments (files) not at fixed positions, since
- segments are simply referenced from a descriptor table
- segments moved between virtual memory, memory, and disk as necessary
- segments can only be 4GB
- segments divided into 4K pages

## Assembly

There are many ways of implementing Assembly language. We'll learn NASM.

The assembly language we use is similar to that of MIPS (from SFWR ENG 3GA3) in terms of its syntax:

%mnemonic (a.k.a. command)% %operator(s)%

Assembly files are saved under the extension .asm

Comment using semicolon ';'

Approach: initially think in pseudo code, then translate it into assembly to make it faster

## Some Commands

Most of the commands are in terms of a **source** and a **destination**. The source can either be an **immediate** (number directly specified in the command), an implied value (such as `i++`, which is

implied to have a source value of 1), or a memory variable. However, the destination must be a variable. Also, **variables** could either refer to a register or memory location. I'll explain what each command means in terms of source and destination so you'll get a better understanding of what each of the terms mean in each context.

Immediates are denoted by the \$ operator

`enter %input variables × 4%, 0: import variables`

- start every program with this
- move ebp so that it's after the space for your input variables
- the first number represents the size taken by the pointers to the variables
  - #pointers to variables × 4 because each pointer takes 4 bytes
- for arrays, you'll need to add 4×#elements
  - Note: the base pointer to any array is the pointer to array[0], which exists no matter what, which counts as an element. So even if it's an empty array, add 4, anyways
- equivalent code: `enter x,0`
  - `push ebp`
  - `move esp, ebp`
  - `sub esp, x`
- `ebp+`

`leave:`

- `counters enter`
- equivalent code:
  - `mov esp, ebp`
  - `pop ebp`

`call %function%:` pauses the next instruction to do the function; it encompasses `enter` and `leave` with `call` and `ret`

`ret:` goes back to `call`, pushes counters `call`

for now, I'm going to represent destination by `d` and source by `s`

`add %d%, %s%:` destination = destination + source

`sub %d%, %s%:` destination = destination – source

`inc %d%:` `d++`

`dec %d%:` `d--`

`mov %d%, %s%:`

- both destination and source cannot be memory locations
- both dest & src must have the same size (# bits)

`mul %d%, %s%`: multiply for unsigned integers  
overflow goes into `edx`

`imul %d%, %s%`: same as `mul`, but for signed integers

`div %s%`: divide for unsigned integers, `eax = eax/s`  
remainder is stored in `edx`

`idiv %s%`: same as `div`, but for signed integers

`cmp %a%, %b%`: flags?: compares the two numbers by doing  $a - b$  and seeing if the zero flag = 1

`test %a%, %b%`: faster than `cmp` for comparing to 0, but other than that, has different flags, so not optimal

`jmp %label%`: jump unconditionally to the specified label (function)

`jxx %option% %label%`: branch if flags?? otherwise continues to next statement

Note that the options for `jxx` don't have a '-', like bash commands

- `near`: (default) jump within segment
  - 32-bit label
- `short`: jump is ?
- `word`: 16-bit label
- `far`: outside of segment
- you will get these on the test

ptr ?

## Stack

First in => last out

`push %s%`: put it into the stack

`pop %d%`: retrieve from the top of the stack

you can only get stuff from the top of the stack

It is useful if you don't have enough registers:

e.g. you need the `eax` and `edx` registers:

`push eax`

`push edx`

`mov eax, 5`

`mov ebx, 3`

`div ebx`

`pop edx`

`pop eax`

It is useful for pointers, sorting, anything where you won't have enough variables.

How the stack looks:

<parameters> ← start at +8

old ebp

<correction/enter heading>

ebp

<local variables>

esp

<anything that is pushed onto the stack>

## Segments/subsections

Some segments you should include are (in order):

- Header: already set in memory, allocated 1000b to do precursors
  - **Precursors:** defines, includes, etc.
  - don't need to specify—it's just everything above all the segments
  - C equivalent: #define
- segment .data: define **data**/initialized variables here
  - put things like prompts here
  - printf formats
    - e.g. 1) format db "%d\n", 0
    - e.g. 2) fobbulous db "%bhaha", 0
  - C equivalent: char \*str = "hi";
- segment .bss: define uninitialized variables here (**b**lock started by **s**ymbol)
  - C equivalent: int n; int[] n;
- segment .text: place the **text** of commands and code and everything else here

e.g.

C: void printf(char[] text, ...);

Assembly:

1. mov eax
2. push format
3. push text
4. call printf
5. add esp, 4\*(vars): ?

## Directives

**directive** assembler and not CPU ?

some applications include:

- define constants
- define memory to store data ?
- group memory into segments
- conditionally include source code ?

- include other files

equ: defines a constant **symbol**, i.e. cannot be re-defined later

syntax: %symbol name% equ %value%

put ‘%’ at the beginning of each line with pre-processor directives, which is different from C, which denotes them with a ‘#’

syntax: %%descriptor% %symbol name% %value%

descriptors:

- include: include other files by replacing the assembled code with the file itself
- define: defines a **symbol** that can be re-defined later

(sorry that I use that symbol for all fields, but it’s more clear than italics, like they do in the text, because I can put spaces, e.g. %any field% and %any% %field% could both be interpreted as: *any field*.)

## Data Directives

**Data directives** (DD) are used in segments to define/reserve room for memory. There are 2 types of data directives:

- resx: **res**erves uninitialized room that is the specified size
- dx: **de**fines/initializes room that is the size of the input

The ‘x’ in each is selected as one of the following letters:

Unit	Letter	Size
byte	b	1B; 4b
word	w	2B; 8b
double word	d	4B; 16b
quad word	q	8B; 32b
ten bytes	t	10B; 40b

Define each directive with a label

syntax: %label% %type% %value%

example: L1 resb 100

(100 uninitialized bytes labeled L1)

Note: the values of each dx can be multiple number systems, so it is necessary to identify which number system you are using, unless you want to initialize it to 0.

Number system	Symbol
Binary	b
Octal	o
Hexadecimal	h

Syntax: %value%%type%

Example: L6 dd 1A92h ; double word initialized to hex 1A92

You can use either “double” or ‘single’ quotes to identify characters/strings of characters.

Faster way to make eax 0 would be:

```
xor  eax,  eax
```

## Types

byte: char, bool, byte

word: short, word

dword: int, float, pointers (32-bit; x86 assembly)

qword: long, dword, pointers (64-bit; x64 assembly)

char[x] = string; array of bytes

## Reliance on a C Driver

Learning to program in assembly helps gain a deeper understanding of how computers work. It also helps better understand compilers and high level languages, such as C.

We'll need a C driver to do many things in assembly, including to:

- call assembly functions
- initialize all segments and registers
- I/O

## Compiling

The driver file can be named anything except for that of your assembly file.

Let's call the driver file driver.c and the assembly file first.asm (because that's the example Ned<sup>2</sup> gave).

1. Compile assembly source: `nasm -f elf first.asm`
  - a. **elf: Executable and Linkable File**
    - i. specifies that elf is the preferred executable type, similar to .exe for Windows
  - b. produces compiled object file: first.o
  - c. `-d ELF_TYPE: ELF_TYPE` is a variable that is similar to define?
2. Compile C driver: `gcc -c driver.c`
  - a. `-c`: compile only
  - b. produces driver.o
  - c. `-m32`: enforces 32-bit code if compiling on 64-bit system
  - d. `-pg`: compile link, profiling
    - i. creates gmon.out
    - ii. run gprof on this executable
    - iii. kprof: application that does profiling
3. Link the 2 compiled object files: `gcc -o first driver.o first.o asm_io.o`
  - a. `-o %output file name%`: gives a name to the output file
    - i. If not specified, default name is: a.out
4. Debug your code by **splicing**, using: `dump_regs %choose random, unique dump number%`.

## Pointers compared to C

```
mov ax, [Data]           ; normal direct memory addressing of a word
mov ebx, Data             ; ebx = & Data
mov ax, [register]        ; ax = *ebx
```

## Recursion

We'll explain it by converting factorial from C to assembly

### C

```
int factorial (int n)
{
    if (n == 1) return 1
    return n * factorial (n-1)
}
```

### Assembly

```
%define n [ebp + 8]
segment .text
    global factorial
factorial:
    enter 0,0
    xor eax, eax ; putting it to 0 (takes 2 bits, whereas mov takes 4 bits)
    xor ebx, ebx ; your compiler actually converts mov to this
    move ax, n
    cmp eax, 1
    je oneCond    ; jump equal
    dec eax
    push eax; n - 1
    call factorial
    pop ecx
    move ax, n
    mul ecx
    jmp end

oneCond: move ax, 1

end:
    mov eax, ecx
    leave
    ret
```

### C

## Variables

static: only in file



local: in block where declared, don't use for recursion, but may take up space otherwise  
global: everywhere in file or any other file (if not static)  
volatile: prevents compiler from optimizing

## Dynamic

```
int *a = malloc(n*sizeof(int))
```

stack grows  
free(a) ← garbage collection, more necessary for global variables

malloc: simply reserves a piece of memory  
calloc: zeroes everything within the given memory  
memset:

## Operators

">%variable%: **pointer**: returns pointer of variable  
&%variable%: **reference**: returns address of variable

## Arrays

Making a pointer array:

```
int a = 1;
int b = 2;

int * pointer[2] = {&a, &b};
printf(*pointer[0]) //1
printf(*pointer[1]) //2
```

**Indirect addressing**: pointers in arrays

## Other stuff

## Personal Questions

Why do you need interrupts when gaming if you know that the person will be gaming? Why don't you just put all the "interrupts" on a different core?

## Shifts

Shifts are in bytes:  
%number% %arrows% %number of bytes%  
int a = 1;  
a << 3; a leftward shift of 3 bytes, so first convert to hex  
10001h  
0008h

## Profiling

**Profiling**: time it takes to run certain segments of your code

Some of them even provide information about memory efficiency (but not all of them)

How to avoid memory leaks: ?

**Code coverage:**

**Loop-unrolling:** reducing number of loops by including more steps in each loop. This allows for better load-sharing over multiple cores

**Locality:**

When you have nested loops processing multi-dimensional arrays, you need to make sure you reference in the correct order because of the way the array is saved depending on how you need to access it.

$$\text{A) } \overbrace{j \left\{ \begin{bmatrix} x_0 & y_0 & z_0 \\ x_1 & y_1 & z_1 \\ x_2 & y_2 & z_2 \end{bmatrix} \right\}}^i \text{ would be saved as } x_0 y_0 z_0 x_1 y_1 z_1 x_2 y_2 z_2$$

$$\text{B) } \begin{bmatrix} x_0 & x_1 & x_2 \\ y_0 & y_1 & y_2 \\ z_0 & z_1 & z_2 \end{bmatrix} \text{ would be saved as } x_0 x_1 x_2 y_0 y_1 y_2 z_0 z_1 z_2$$

If you're trying to access each coordinate set, it's better to choose A because the 3 data locations are beside each other.

If you're trying to compare each component or something else that is one component at a time, it's better to choose B.

In general: for  $A[i][j]$ , have `for i` as the outer loop and `for j` as the inner loop

**Pipelining:** i know it but don't know how to use it ?

is it just basically that you mix up your computations even if it results in less readable code to allow the pipelining to be optimized?

**Avoid division:** avoids having to deal with overflow

`if (a/b > c)` replace by  
`if (a > c * b)`