IRM LAB RULES and APPENDIX

Introduction to Robotics and Mechatronics

FS 2022

Contents

Appendix A

Laboratory Rules

Author:

Alexandre Mesot, Jonas Lussi, Naveen Shamsudhin and Prof. Bradley J. Nelson

Multiscale Robotics Lab, Institute of Robotics and Intelligent Systems

Date: 2022

Version: 1.2

A.1 Corona related Laboratory Rules

Due to the restrictions caused by COVID-19 you will have to follow these rules

- During your laboratory work you have to carefully respect the ETH guidelines on COVID-19

A.2 General Laboratory Rules

In this lab, you will use and program an embedded computing platform, and interface several sensors and actuators to develop a feedback control system. For a smooth running of the laboratory sessions, you are obliged to read the following rules and adhere to them over the duration of the course.

- If you are unsure of electrical connections, please ask the lab assistants before proceeding.
- Connect cables carefully - do not pull or use excessive force on them.
- Be attentive and walk around the room carefully to avoid disturbing the lab hardware and cabling.
- That being said, accidents happen. Please report any damaged or non-functioning equipment.
- If lab material is missing, damaged or broken due to mishandling, all students will lose access to the exercise room outside regular lab hours.

Appendix B

Operating System

Author: Daniel Steinmann

Version: 1

B.1 Virtual Machine

In this course, we will work with Ubuntu 18.04. We *highly recommend* you to use a virtual machine and the provided image that already contains a preinstalled environment with the following software:

- gcc & g++ build essentials (Command-line compiler)
- SublimeText 3 (Customizeable editor)
- Arduino IDE (With installed drivers for Featherboard)
- git (Source control, optional)
- OpenCV 3.6

Install VMWare

To run the provided image you need the VMWare Workstation 16 Pro (Windows, Linux) or VMWare Fusion 12 (macOS). This software can be ordered on the ETH Zurich IT Shop (free for students): `https://itshop.ethz.ch/`, search for "VMware Academic Program".

Follow the instructions to download and install the software. We recommend you to have at least 20GB of available disk space on your computer to run the virtual machine.

Download Image

Download the zipped folder from Polybox (ca. 5 GB, 10GB unzipped) and extract it to the directory where you want the virtual machine to be stored.
`https://polybox.ethz.ch/index.php/s/V0BIgEat7JZ69r0`

Start Up Virtual Machine

- Open VMWare Workstation
- Open file IRM_Ubuntu.vmx in the downloaded folder

- Start the virtual machine with *Power on this virtual machine* or *Start up this guest operating system*. A prompt will open and ask if you copied or moved the virtual machine, click *I copied it.*.
- **Important:** If it is the first time you are using a virtual machine on your laptop, there might be an error message that tells you that “This host supports Intel VT-x, but Intel VT-x is disabled”. The Intel Virtual Technology (Intel VT)has to be enabled in your BIOS (or UEFI). You will have to restart the computer and press either Enter, F1, F10, or DEL to go to BIOS settings (depending on your PC manufacturer). Under Security->System Security you will find the option to enable VT. Some more explanations can be found here.
- If you are prompted to install *VMWare Tools for Linux*, click on *Donwload and Install*.
- To log in, use the provided user and password:
  - User: irm
  - Password: msrl
- Initially you might see a small window size. Minimizing or maximizing the *VMWare*-Window will cause the virtual machine desktop to resize.
- The default setting is a US-Keyboard. To change this, change the keyboard language using the dropdown-menu in the upper right corner of the desktop. If some letters and symbols are not located at the same location as on your keyboard go to the Settings application, go to the Region & Language section and add the input source that best matches your keyboard layout (for example `German (Switzerland, Macintosh)` for Swiss-German layout Macs).

Serial Connection

To communicate with the microcontroller, the virtual machine needs access to the corresponding serial port. In order to connect the microcontroller, follow these steps:

- Connect the microcontroller to your computer via USB-cable.
- Start up the virtual machine.
- On the upper left corner of VMware, click on the tab *VM* and navigate to *Removable Devices*.
- Find the entry for *Silicon CP2104 USB to UART Bridge Controller*.
  Click on *Connect (Disconnect from host)*.
- Check the connection by opening the terminal and entering the following command: `dmesg | grep tty`
- The command line should return a message confirming that a device is attached to port `ttyUSB0`.

B.2 Using your own Operating System

Depending on your operating system, it is also possible to install the necessary resources on your own computer without a virtual machine. While this will save hardware resources, we can not guarantee that the provided code runs on your system, and we cannot provide support for individual operating system solutions. For those who like the challenge, find here a list of necessary software components:

- C Compiler (For example Cygwin+MinGW on Windows)
- Arduino IDE (Follow the instructions here to install the Featherboard drivers)
- Serial communcation: Since naming of the serial port and the used protocols vary by operating system, you will need to implement your own serial communication functions. There are many resources available online, but it might take some time to find the right configuration.

Appendix C

Programming Reference

C.1 Unix Commands

Author: Peter Burden

Edited by Brad Kratochvil, Kathrin Peyer

Version: 0.3

Unix Commands

- `Using the System`
- `Basic File Manipulations`
- `File Naming Conventions`
- `Use of Wildcards`
- `Escaping Conventions`

Using the system Unix systems are normally driven from the **command line**. This means that if you want the system to do something such as start an application or manipulate some files you will need to type in from the keyboard an appropriate command, usually with one or more parameters.

This user supplied input is in response to a **command interpreter** or **shell** prompt.

On a typical Unix system there are several hundred standard commands that come "with the system". Many system administrators install extra software making further commands available. There is no simple single list of the available commands. Some commands give a brief summary of available options if they are issued without any options or with faulty options but it is much better to consult the on-line manual. If you are using X-windows, you will still need to use the command line interface, you'll do this via a command tool or command window.

The behaviour of Unix commands is usually controlled by command line options which are identified by a single letter preceded by a dash (**not** a slash as in MSDOS). As with DOS all user command input is actually input to a command interpreter program. This is called **the shell** and may be one of several well known programs such as.

Name	Path	Comments
Bourne shell	/bin/sh	The original. Highly standardized but lacking in some facilities. Still widely used for scripts intended to be portable. Named for the person who wrote it.
The C shell	/bin/csh	Now obsolescent but once widely used especially by devotes of the BSD flavours of Unix. Very different from the Bourne shell.
The Bash shell	/usr/local/bin/bash	An enhanced variant of the Bourne shell from the GNU Free Software Foundation. The name derives from the punning "Born Again Shell".
The Korn shell	/bin/ksh	A supposed replacement for the Bourne and C shells that never really caught on. Lots of powerful facilities.

The DOS command interpreter is called, of course, COMMAND.COM. There is no reason why under both Unix and MSDOS you shouldn't write or install your own command interpreter. Unix-like command interpreters for MSDOS are quite common, nobody has ever written a version of COMMAND.COM for Unix.

Basic File Manipulations There are many Unix commands for manipulating files. The following list presents some of the more basic commands. Many of these commands have extensive option sets.

Command	Function	MSDOS Equivalent
cp	Copy a file or collection of files	COPY
mv	Move or rename a file or collection of files	COPY REN
rm	Remove a file or collection of files	DEL
ln	Create a link (or alias) from one file to another	No equivalent
ls	List a directory. There are a **lot** of options	DIR
mkdir	Create a directory	MKDIR
rmdir	Remove a directory (must be empty)	RMDIR

All commands understand "wild-card" conventions which are discussed later.

Naming Files The Unix conventions for naming files are very simple and are summarised below.

- A file name can be up to 255 characters long. This restriction is a **file name** restriction **not** a path name length restriction.
- A file name can contain any character at all with the exception of slash ("/") which is used as a path name component separator and NUL (a character with all bits set to zero) which is used internally as a file name terminator. Thus a file name could consist entirely of space and back space characters.
- All characters are significant. In other words the Unix file naming system is case sensitive. Lower case file names are more common than upper case file names.
- File names do not have components or extensions. You can have as many dots as you like (up to 255 !) in a Unix file name. Many applications, however, regard the part of a file name after the last dot as having special significance.
- Any freshly created directory will contain entries "." and ".." (*dot* and *dot-dot* referring to the current directory and the parent directory, thus files cannot be named "." and ".."

Wildcards Wildcards are used to enable several files with related names to be manipulated by a single command. There is a very important difference between the handling of wildcards by DOS systems and by Unix systems.

On DOS systems when you issue a command such as **DEL** $*$**.pas** to delete all the files in the current directory which have the extension **.pas** the character sequence $*$**.pas** is passed unchanged to the DEL command which includes the necessary code to examine the directory and identify the relevant files. This means that if you want to write your own programs to manipulate sets of files identified by wildcards you will have to write the relevant code, although most DOS compilers will provide library routines to do most of the work.

On a Unix system it is the responsibility of tbe command interpreter (or shell) to examine the user input and convert the wild card expression to a list of file names which are then passed to the selected command. This means that wildcard expansion is doen cosistently, it also means that if you want to write your own programs that will work with wildcards all you need to do is to expect a list of file names,

The following wildcards are understood by all the normal Unix shells.

*	Matches any sequence of characters
?	Matches any single character
[..]	Matches any of the characters included in the list between the brackets. **Ranges** may also be specified using a dash so, for example, **[a-z]** means any lower case letter.
[!..]	Matches any character **not** included in the list

The shell operates by considering all the file names in the directory in turn and checking whether they match the wild-card expression. In performing this determination the shell regards . (*dot*) as a normal character.

Here are some examples of file names and wild-card expressions and an indication of whether they match

File Names	Wild-card Expressions				
	*.c	*c	a*c	*[0-9]*c	[0-9]*
myprog.c	Y	Y	N	N	N
assess1.c	Y	Y	Y	Y	N
artic	N	Y	Y	N	N
013.c	Y	Y	N	Y	Y
01357	N	N	N	N	Y
x.y.z.c	Y	Y	N	N	N
access.artic	N	Y	Y	N	N

Escaping Conventions Since Unix allows arbitrary file names and since the command interpretive shell understands various wild-card characters there are difficulties handling files with "unusual" names. In general file names containing any non-printing character (including new-line and space) and any of the following characters will cause problems.

Character	Normal Use
*	File naming wild card
?	File naming wild card
[	File naming wild card
]	File naming wild card
$	Shell variable substitution
&	Shell backgrounding
<	Input redirection
>	Output redirection
|	Shell piping
(	Shell command grouping
)	Shell command grouping
{	Shell variable substitution
}	Shell variable substitution
;	Shell command grouping
\	Shell escaping
"	Shell escaping
’	Shell escaping
‘	Shell command substitution

All the above should normally be avoided in file naming. The characters "^" and "!" may also cause problems. File names starting with "-" can also cause difficulties.

All the above apply to file names entered via the shell command interpreter. No restrictions apply to file names used within programs.

It is possible to overcome many of the constraints by **quoting** or

escaping file names containing awkward characters. There are several ways of doing this and for full discussion of all the details and ramifications the manual pages for your command interpretive shell should be consulted. Basically file names can be enclosed in either single or double quotes or the individual awkward characters preceded by a back-slash character (\).

The following examples show how to delete files with "awkward" characters in their names.

File Name	Command to delete	Comments
*	rm "*"	`rm *` would also delete the file but this would also have undesirable side-effects.
my file	rm "my file"	`rm my file` would result in the deletion of files called "my" and "file".
$prices	rm $prices	Failure to escape would result in the deletion of a file whose name was stored in the shell variable "prices"

C.2 C Program Compilation

Author: Peter Burden

Edited by Brad Kratochvil

Version: 0.2

C Compilation under Unix

- `The Source File`
- `Compilation`
- `Running the compiled program`

Source File The first step is to create a source file using a text editor such as vi, emacs, or textedit. The source file name should end with the characters ".c"

Compilation Assuming the source file is called *myprog.c* it can be compiled using the command cc or, if preferred gcc. Remember that these are C compilers **not** C++ compilers so make sure you understand the difference.

Use the command thus

```
cc myprog.c
```

This will, if there are no compilation errors, create a file called *a.out* which contains the executable version of the program.

If there are compilation errors you'll need to edit the source file and have another go. You'll find it particularly convenient, when working under X-windows, to open a separate textedit window. You can then edit the source code in the textedit window, save the results without closing the window, issue the compilation command in a command window and observe the results in the command window at the same time as viewing the source in the textedit window.

If you want to capture the compiler error messages to a file for printing out, you'll need to remember that the error messages go to *stderr*

and not *stdout*. Capture the error messages thus

```
cc myprog.c 2>myprog.errs
```

If you want a specific output file you can use the following version of the *cc* command

```
cc -o myprog myprog.c
```

which creates the executable code in the file called *myprog*.

If you want to incorporate routines from special libraries you'll need further command line options thus

```
cc -c myprog myprog.c -lm -ls626
```

which uses the math and Sensoray libraries.

Running the program The executable program can be run by simply typing the name of the file at the system command prompt.

The only possible complication would be if there were another program of the same name in one of the standard directories, if you have decided to call your program *chmod* then typing *chmod* will run the system command *chmod*, not your program.

To be certain you are executing the program you just compiled, put the path in front of the program name. Assuming that your user account is joeuser and you have compiled the program in your root directory, the following commands are all equivalent.

```
/home/joeuser/myprog
~/myprog
./myprog
```

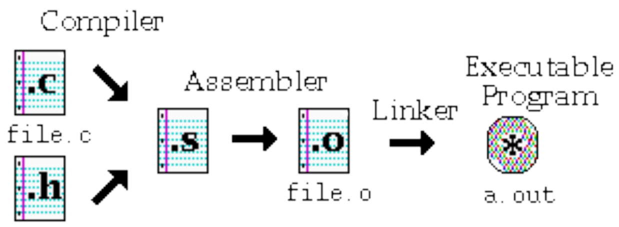


Figure C.1: A simple compilation

C.3 Makefiles

Author: Ben Y. Yoshino

University of Hawaii, College of Engineering

Edited by Brad Kratochvil

Date: 1995

Version: 0.2

C.3.1 Makefiles

The make command allows you to manage large programs or groups of programs. As you begin to write larger programs, you will notice that re-compiling larger programs takes much longer than re-compiling short programs. Moreover, you notice that you usually only work on a small section of the program (such as a single function that you are debugging), and much of the rest of the program remains unchanged.

The make program aids you in developing your large programs by keeping track of which portions of the entire program have been changed, compiling only those parts of the program which have changed since the last compile.

C.3.2 A simple compilation

Compiling a small C program requires at least a single .c file, with .h files as appropriate. Although the command to perform this task is simply cc file.c, there are 3 steps to obtain the final executable program, as shown:

1. Compiler stage: All C language code in the .c file is converted into a lower-level language called Assembly language; making .s files.
2. Assembler stage: The assembly language code made by the previous stage is then converted into object code which are fragments of code which the computer understands directly. An object code file ends with .o.
3. Linker stage: The final stage in compiling a program involves linking the object code to code libraries which contain certain "built-in" functions, such as printf. This stage produces an executable program, which is named a.out by default.

C.3.2.1 Compiling with several files

When your program becomes very large, it makes sense to divide your source code into separate easily-manageable .c files. The figure above demonstrates the compiling of a program made up of two .c files and a single common.h file. The command is as follows:

```
cc green.c blue.c
```

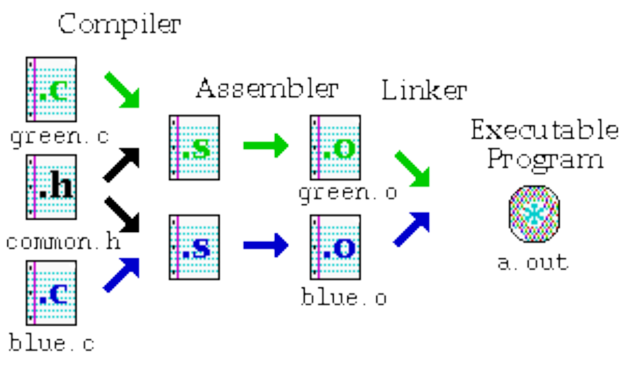


Figure C.2: Compiling with several files

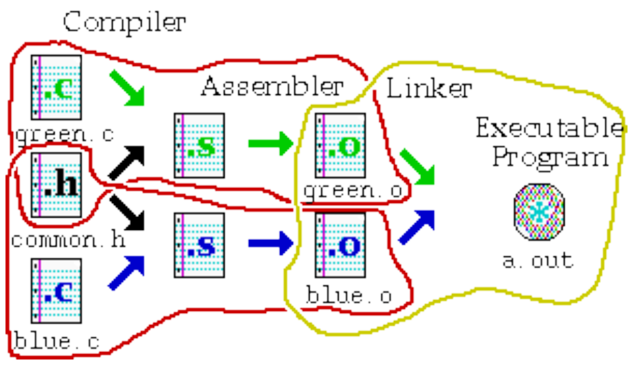


Figure C.3: Separate compilation

where both .c files are given to the compiler. Note that the first two steps taken in compiling the files are identical to the previous procedure for a single .c file, but the last step has an interesting twist: The two .o files are linked together at the Linker stage to create one executable program, a.out.

C.3.2.2 Separate compilation

The steps taken in creating the executable program can be divided up in to two compiler/assembler steps circled in red, and one final linker step circled in yellow. The two .o files may be created separately, but both are required at the last step to create the executable program.

You can use the -c option with cc to create the corresponding object (.o) file from a .c file. For example, typing the command: cc -c green.c will not produce an a.out file, but the compiler will stop after the assembler stage, leaving you with a green.o file.

C.3.2.3 Separate compilation steps

The three different tasks required to produce the executable program are as follows:

- Compile green.o: cc -c green.c
- Compile blue.o: cc -c blue.c
- Link the parts together: cc green.o blue.o

For example, it is important to note that in order to create the file, green.o, the two files, green.c and the header file common.h are required. Similarly, in order to create the executable program, a.out, the object files green.o and blue.o are required.

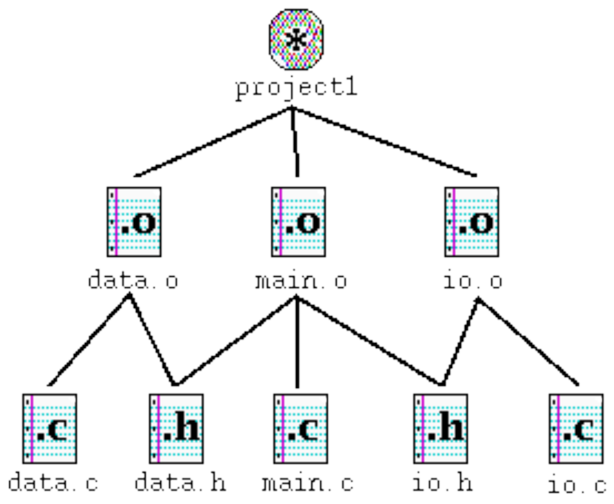


Figure C.4: Dependency graphs

C.3.2.4 Splitting your C program

When you separate your C program into many files, keep these points in mind:

- Be sure no two files have functions with the same name in it. The compiler will get confused.
- Similarly, if you use global variables in your program, be sure no two files define the same global variables.
- If you use global variables, be sure only one of the files defines them, and declare them in your .h as follows: extern int globalvar;
- To use functions from another file, make a .h file with the function prototypes, and use #include to include those .h files within your .c files.
- At least one of the files must have a main() function.

Note: When you define a variable, it looks like this: int globalvar;. When you declare a variable, it looks like this: extern int globalvar;. The main difference is that a variable definition creates the variable, while a declaration indicates that the variable is defined elsewhere. A definition implies a declaration.

C.3.2.5 Dependencies

The principle by which make operates was described to you in the last section. It creates programs according to the file dependencies. For example, we now know that in order to create an object file, program.o, we require at least the file program.c. (There may be other dependencies, such as a .h file.)

This section involves drawing what are called "@ref dep_graph", which are very similar to the diagrams given in the previous section. As you become proficient using make, you probably will not need to draw these diagrams, but it is important to get a feel for what you are doing.

C.3.2.6 Dependency graphs

This graph shown in the figure is a program which is made up of 5 source files, called data.c, data.h, io.c, io.h, and main.c. At the top is the final result, a program called project1. The lines which radiate downwards from a file are the other files which it depends on. For example, to create main.o, the three files data.h, io.h, and main.c are needed.

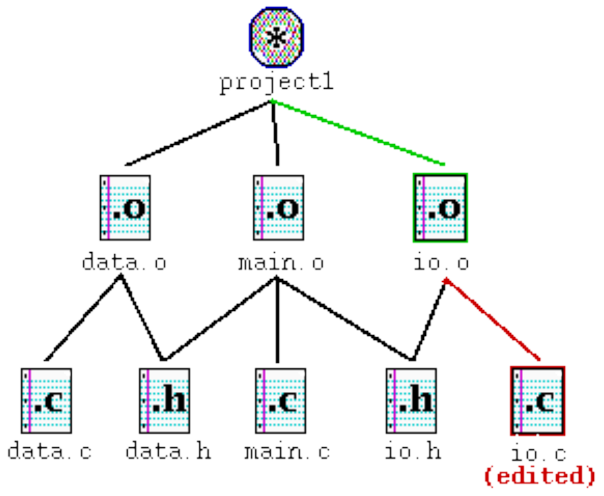
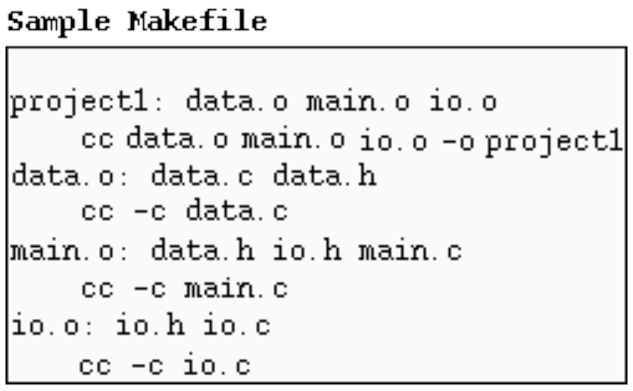


Figure C.5: How dependency works



Sample Makefile

```
project1: data.o main.o io.o
    cc data.o main.o io.o -o project1
data.o: data.c data.h
    cc -c data.c
main.o: data.h io.h main.c
    cc -c main.c
io.o: io.h io.c
    cc -c io.c
```

Figure C.6: Sample Makefile

Suppose that you have gone through the process of compiling the program, and while you are testing the program, you realize that one function in io.c has a bug in it. You edit io.c to fix the bug.

The figure above shows io.c outlined in red. By going up the graph, you notice that io.o needs to be updated because io.c has changed. Similarly, because io.o has changed, project1 needs to be updated as well.

The make program gets its dependency "graph" from a text file called makefile or Makefile which resides in the same directory as the source files. Make checks the modification times of the files, and whenever a file becomes "newer" than something that depends on it, (in other words, modified) it runs the compiler accordingly.

For example, the previous page explained io.c was changed. If you edit io.c, it becomes "newer" than io.o, meaning that make must run cc -c io.c to create a new io.o, then run cc data.o main.o io.o -o project1 for project1.

C.3.3 The Makefile

The previous section described dependencies between files. This section describes the make program in more detail by describing the file it uses, called makefile or Makefile. This file determines the relationships between the source, object and executable files.

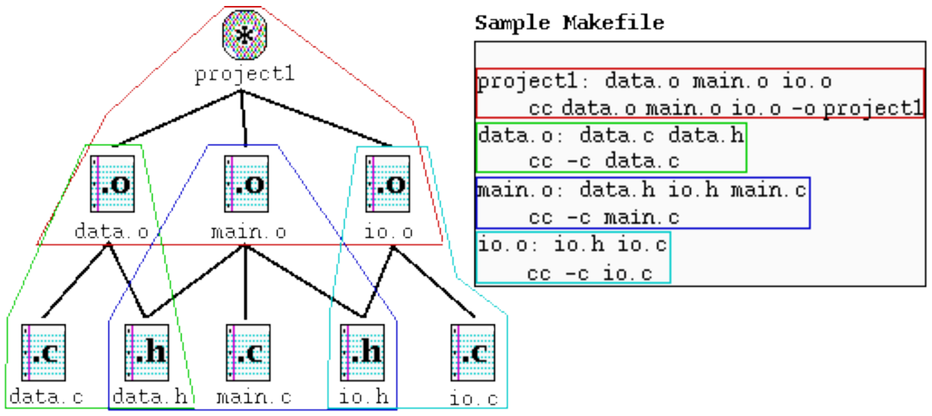


Figure C.7: Translating the dependency graph

C.3.3.1 Translating the dependency graph

Each dependency shown in the graph is circled with a corresponding color in the Makefile, and each uses the following format:

```
target : source file(s)
    command (must be preceded by a tab)
```

A target given in the Makefile is a file which will be created or updated when any of its source files are modified. The command(s) given in the subsequent line(s) (which must be preceded by a tab character) are executed in order to create the target file.

C.3.3.2 Listing Dependencies

```
project1: data.o main.o io.o
    cc data.o main.o io.o -o project1
data.o: data.c data.h
    cc -c data.c
main.o: data.h io.h main.c
    cc -c main.c
io.o: io.h io.c
    cc -c io.c
```

Note that in the Makefile shown above, the .h files are listed, but there are no references in their corresponding commands. This is because the .h files are referred within the corresponding .c files through the #include "file.h". If you do not explicitly include these in your Makefile, your program will not be updated if you make a change to your header (.h) files.

Attention: Comments can be placed in a Makefile by placing a pound sign (#) in front of it.

C.3.3.3 Using the Makefile with make

Once you have created your Makefile and your corresponding source files, you are ready to use make. If you have named your Makefile either Makefile or makefile, make will recognize it. If you do not wish to call your Makefile one of these names, you can use make -f mymakefile. The order in which dependencies are listed is important. If you simply type make and then return, make will attempt to create or update the first dependency listed.

You can also specify one of the other targets listed in the Makefile, and only that target (and its corresponding source files) would be made. For example, if we typed make, the output of make would look as follows:

```
% make
    cc -c data.c
    cc -c main.c
    cc -c io.c
    cc data.o main.o io.o -o project1
```

When making its targets, make first checks the source files and attempts to create or update the source files. That is why data.o, main.o and io.o were created before attempting to create the target: project1.

C.3.4 Shortcuts for make

The make program has many other features which have not been discussed in previous sections. Most important of these features is the macro feature. Macros in make work similarly to macros used in C programming. Make also has its own pre-defined rules which you can take advantage of to make your Makefile smaller.

C.3.4.1 Macros in make

The make program allows you to use macros, which are similar to variables, to store names of files. The format is as follows: OBJECTS = data.o io.o main.o Whenever you want to have make expand these macros out when it runs, type the following corresponding string .

Here is our sample Makefile again, using a macro.

```
OBJECTS = data.o main.o io.o
project1: $(OBJECTS)
    cc $(OBJECTS) -o project1
data.o: data.c data.h
    cc -c data.c
main.o: data.h io.h main.c
    cc -c main.c
io.o: io.h io.c
    cc -c io.c
```

You can also specify a macro's value when running make, as follows: make 'OBJECTS=data.o newio.o main.o' project1 This overrides the value of OBJECTS in THE Makefile

C.3.4.2 Special macros

In addition to those macros which you can create yourself, there are a few macros which are used internally by the make program. Here are some of those, listed below:

CC

- Contains the current C compiler. Defaults to cc.

CFLAGS

- Special options which are added to the built-in C rule. (See next page.)

$@

- Full name of the current target.

$?

- A list of files for current dependency which are out-of-date.

$<

- The source file of the current (single) dependency.

You can also manipulate the way these macros are evaluated, as follows, assuming that OBJS = data.o io.o main.o, using $(OBJS:.o=.c) within the Makefile substitutes .o at the end with .c, giving you the following result: data.c io.c main.c

C.3.4.3 Predefined rules

By itself, make knows already that in order to create a .o file, it must use cc -c on the corresponding .c file. These rules are built into make, and you can take advantage of this to shorten your Makefile. If you just indicate the .h files in the dependency line of the Makefile that the current target is dependent on, make will know that the corresponding .c file is already required. You don't even need to include the command for the compiler.

This reduces our Makefile further, as shown:

```
OBJECTS = data.o main.o io.o
project1: $(OBJECTS)
        cc $(OBJECTS) -o project1
data.o: data.h
main.o: data.h io.h
io.o: io.h
```

One thing to consider, however, is that when you are compiling programs on Wiliki, you may wish to add a CFLAGS macro at the top of your Makefile to enable the compiler to use ANSI standard C compilation. The macro looks like this: CFLAGS=-Aa -D_HPUX_SOURCE This will allow make to use ANSI C with the predefined rules.

C.3.4.4 Miscellaneous shortcuts

Although the examples we have shown do not explicitly say so, you can put more than one file in the target section of the dependency rules. If a file appears as a target more than once in a dependency, all of its source files are included as sources for that target.

Here is our sample Makefile again:

```
CFLAGS = -Aa -D_HPUX_SOURCE
OBJECTS = data.o main.o io.o
project1: $(OBJECTS)
        cc $(OBJECTS) -o project1
data.o main.o: data.h
io.o main.o: io.h
```

This Makefile shows main.o appearing in two places. Make knows by looking at all the dependencies that main.o depends on both data.h and io.h.

C.3.5 Literature Reference

The remainder of this article can be found at `http://www.eng.hawaii.edu/Tutor/Make/`.

C.4 What is a Pointer?

Author: Ted Jensen

Edited by Brad Kratochvil

Date: February 2000

Version: 1.2

C.4.1 What is a pointer?

One of those things beginners in C find difficult is the concept of pointers. The purpose of this tutorial is to provide an introduction to pointers and their use to these beginners. I have found that often the main reason beginners have a problem with pointers is that they have a weak or minimal feeling for variables, (as they are used in C). Thus we start with a discussion of C variables in general.

A variable in a program is something with a name, the value of which can vary. The way the compiler and linker handles this is that it assigns a specific block of memory within the computer to hold the value of that variable. The size of that block depends on the range over which the variable is allowed to vary. For example, on 32 bit PC's the size of an integer variable is 4 bytes. On older 16 bit PCs integers were 2 bytes. In C the size of a variable type such as an integer need not be the same on all types of machines. Further more there is more than one type of integer variable in C. We have integers, long integers and short integers which you can read up on in any basic text on C. This document assumes the use of a 32 bit system with 4 byte integers.

If you want to know the size of the various types of integers on your system, running the following code will give you that information.

```
#include <stdio.h>

int main()
{
printf("size of a short is %d\n", sizeof(short));
printf("size of a int is %d\n", sizeof(int));
printf("size of a long is %d\n", sizeof(long));
}
```

If determining the number of bits to be used is desired, the following definitions can alternatively be used.

Type Declaration	# of Bits	Range
intN_t	N-bit, N $\in$ {8,16,32,64}	$[-2^{N-1}, +2^{N}-1]$
uintN_t	N-bit, N $\in$ {8,16,32,64}	$[0, 2^{N}-1]$

With these, the number of bits in a variable is explicitly determined. One has to be very careful when selecting the type and the number of bits to use, keeping the limitations always in mind. For instance, an 8 bit unsigned integer can hold values in $[0, 255]$ while an 8 bit integer can hold values in $[-128, +127]$. So, if a variable is expected to attain positive values greater than 128, the former type should be preferred since the later does not have the capacity.

When we declare a variable we inform the compiler of two things, the name of the variable and the type of the variable. For example, we declare a variable of type integer with the name k by writing:

```
    int k;
```

On seeing the "int" part of this statement the compiler sets aside 4 bytes of memory (on a PC) to hold the value of the integer. It also sets up a symbol table. In that table it adds the symbol k and the relative address in memory where those 4 bytes were set aside.

Thus, later if we write:

```
    k = 2;
```

we expect that, at run time when this statement is executed, the value 2 will be placed in that memory location reserved for the storage of the value of k. In C we refer to a variable such as the integer k as an "object".

In a sense there are two "values" associated with the object k. One is the value of the integer stored there (2 in the above example) and the other the "value" of the memory location, i.e., the address of k. Some texts refer to these two values with the nomenclature rvalue (right value, pronounced "are value") and lvalue (left value, pronounced "el value") respectively.

In some languages, the lvalue is the value permitted on the left side of the assignment operator '=' (i.e. the address where the result of evaluation of the right side ends up). The rvalue is that which is on the right side of the assignment statement, the 2 above. Rvalues cannot be used on the left side of the assignment statement. Thus: 2 = k; is illegal.

Actually, the above definition of "lvalue" is somewhat modified for C. According to K&R II (page 197): [1]

"An object is a named region of storage; an lvalue is an expression referring to an object."

However, at this point, the definition originally cited above is sufficient. As we become more familiar with pointers we will go into more detail on this.

Okay, now consider:

```
   int j, k;

    k = 2;
    j = 7;    <-- line 1
    k = j;    <-- line 2
```

In the above, the compiler interprets the j in line 1 as the address of the variable j (its lvalue) and creates code to copy the value 7 to that address. In line 2, however, the j is interpreted as its rvalue (since it is on the right hand side of the assignment operator '='). That is, here the j refers to the value stored at the memory location set aside for j, in this case 7. So, the 7 is copied to the address designated by the lvalue of k.

In all of these examples, we are using 4 byte integers so all copying of rvalues from one storage location to the other is done by copying 4 bytes. Had we been using two byte integers, we would be copying 2 bytes.

Now, let's say that we have a reason for wanting a variable designed to hold an lvalue (an address). The size required to hold such a value depends on the system. On older desk top computers with 64K of memory total, the address of any point in memory can be contained in 2 bytes. Computers with more memory would require more bytes to hold an address. The actual size required is not too important so long as we have a way of informing the compiler that what we want to store is an address.

Such a variable is called a pointer variable (for reasons which hopefully will become clearer a little later). In C when we define a pointer variable we do so by preceding its name with an asterisk. In C we also give our pointer a type which, in this case, refers to the type of data stored at the address we will be storing in our pointer. For example, consider the variable declaration:

```
   int *ptr;
```

Null pointer

ptr is the name of our variable (just as k was the name of our integer variable). The '*' informs the compiler that we want a pointer variable, i.e. to set aside however many bytes is required to store an address in memory. The int says that we intend to use our pointer variable to store the address of an integer. Such a pointer is said to "point to" an integer. However, note that when we wrote int k; we did not give k a value. If this definition is made outside of any function ANSI compliant compilers will initialize it to zero. Similarly, ptr has no value, that is we haven't stored an address in it in the above declaration. In this case, again if the declaration is outside of any function, it is initialized to a value guaranteed in such a way that it is guaranteed to not point to any C object or function. A pointer initialized in this manner is called a "null" pointer.

The actual bit pattern used for a null pointer may or may not evaluate to zero since it depends on the specific system on which the code is developed. To make the source code compatible between various compilers on various systems, a macro is used to represent a null pointer. That macro goes under the name NULL. Thus, setting the value of a pointer using the NULL macro, as with an assignment statement such as ptr = NULL, guarantees that the pointer has become a null pointer. Similarly, just as one can test for an integer value of zero, as in if(k == 0), we can test for a null pointer using if (ptr == NULL).

But, back to using our new variable ptr. Suppose now that we want to store in ptr the address of our integer variable k. To do this we use the unary & operator and write:

```
    ptr = &k;
```

What the & operator does is retrieve the lvalue (address) of k, even though k is on the right hand side of the assignment operator '=', and copies that to the contents of our pointer ptr. Now, ptr is said to "point to" k. Bear with us now, there is only one more operator we need to discuss.

The "dereferencing operator" is the asterisk and it is used as follows:

```
    *ptr = 7;
```

will copy 7 to the address pointed to by ptr. Thus if ptr "points to" (contains the address of) k, the above statement will set the value of k to 7. That is, when we use the '*' this way we are referring to the value of that which ptr is pointing to, not the value of the pointer itself.

Similarly, we could write:

```
 printf("%d\n",*ptr);
```

to print to the screen the integer value stored at the address pointed to by ptr.

One way to see how all this stuff fits together would be to run the following program and then review the code and the output carefully.

```
------------ Program 1.1 ---------------------------------

/* Program 1.1 from PTRTUT10.TXT   6/10/97 */

#include <stdio.h>

int j, k;
int *ptr;

int main(void)
{
    j = 1;
    k = 2;
    ptr = &k;
    printf("\n");
    printf("j has the value %d and is stored at %p\n", j, (void *)&j);
    printf("k has the value %d and is stored at %p\n", k, (void *)&k);
    printf("ptr has the value %p and is stored at %p\n", ptr, (void *)&ptr);
    printf("The value of the integer pointed to by ptr is %d\n", *ptr);

    return 0;
}
```

Note: We have yet to discuss those aspects of C which require the use of the (void *) expression used here. For now, include it in your test code. We'll explain the reason behind this expression later.

To review:

- A variable is declared by giving it a type and a name (e.g. int k;)
- A pointer variable is declared by giving it a type and a name (e.g. int ∗ptr) where the asterisk tells the compiler that the variable named ptr is a pointer variable and the type tells the compiler what type the pointer is to point to (integer in this case).
- Once a variable is declared, we can get its address by preceding its name with the unary & operator, as in &k.
- We can "dereference" a pointer, i.e. refer to the value of that which it points to, by using the unary '∗' operator as in ∗ptr.
- An "lvalue" of a variable is the value of its address, i.e. where it is stored in memory. The "rvalue" of a variable is the value stored in that variable (at that address).

For more information, see Pointers and Structures.

C.4.2 References

1. B. Kernighan and D. Ritchie, *The C Programming Language* 2nd Edition, Prentice Hall. ISBN 0-13-110362-8

C.5 Pointers and Structures

Author: Ted Jensen

Edited by Brad Kratochvil,

Date: February 2000

Version: 1.2

C.5.1 Structures

As you may know, we can declare the form of a block of data containing different data types by means of a structure declaration. For example, a personnel file might contain structures which look something like:

```
    struct tag {
        char lname[20];        /* last name */
        char fname[20];        /* first name */
        int age;               /* age */
        float rate;            /* e.g. 12.75 per hour */
    };
```

Let's say we have a bunch of these structures in a disk file and we want to read each one out and print out the first and last name of each one so that we can have a list of the people in our files. The remaining information will not be printed out. We will want to do this printing with a function call and pass to that function a pointer to the structure at hand. For demonstration purposes I will use only one structure for now. But realize the goal is the writing of the function, not the reading of the file which, presumably, we know how to do.

For review, recall that we can access structure members with the dot operator as in:

```
--------------- program 5.1 ------------------

/* Program 5.1 from PTRTUT10.HTM     6/13/97 */


#include <stdio.h>
#include <string.h>

struct tag {
    char lname[20];      /* last name */
    char fname[20];      /* first name */
    int age;             /* age */
    float rate;          /* e.g. 12.75 per hour */
};

struct tag my_struct;       /* declare the structure my_struct */

int main(void)
{
    strcpy(my_struct.lname,"Jensen");
    strcpy(my_struct.fname,"Ted");
    printf("\n%s ",my_struct.fname);
    printf("%s\n",my_struct.lname);
    return 0;
}

-------------- end of program 5.1 --------------
```

Now, this particular structure is rather small compared to many used in C programs. To the above we might want to add:

```
    date_of_hire;                  (data types not shown)
    date_of_last_raise;
    last_percent_increase;
    emergency_phone;
    medical_plan;
    Social_S_Nbr;
    etc.....
```

If we have a large number of employees, what we want to do is manipulate the data in these structures by means of functions. For example we might want a function print out the name of the employee listed in any structure passed to it. However, in the original C (Kernighan & Ritchie, 1st Edition) it was not possible to pass a structure, only a pointer to a structure could be passed. In ANSI C, it is now permissible to pass the complete structure. But, since our goal here is to learn more about pointers, we won't pursue that.

Anyway, if we pass the whole structure it means that we must copy the contents of the structure from the calling function to the called function. In systems using stacks, this is done by pushing the contents of the structure on the stack. With large structures this could prove to be a problem. However, passing a pointer uses a minimum amount of stack space.

In any case, since this is a discussion of pointers, we will discuss how we go about passing a pointer to a structure and then using it within the function.

Consider the case described, i.e. we want a function that will accept as a parameter a pointer to a structure and from within that function we want to access members of the structure. For example we want to print out the name of the employee in our example structure.

Okay, so we know that our pointer is going to point to a structure declared using struct tag. We declare such a pointer with the declaration:

```
    struct tag *st_ptr;
```

and we point it to our example structure with:

```
    st_ptr = &my_struct;
```

Now, we can access a given member by de-referencing the pointer. But, how do we de-reference the pointer to a structure? Well, consider the fact that we might want to use the pointer to set the age of the employee. We would write:

```
    (*st_ptr).age = 63;
```

Look at this carefully. It says, replace that within the parenthesis with that which st_ptr points to, which is the structure my_struct. Thus, this breaks down to the same as my_struct.age.

However, this is a fairly often used expression and the designers of C have created an alternate syntax with the same meaning which is:

```
    st_ptr->age = 63;
```

With that in mind, look at the following program:

```
------------ program 5.2 ---------------------

/* Program 5.2 from PTRTUT10.HTM   6/13/97 */
```

```
#include <stdio.h>
#include <string.h>

struct tag{                     /* the structure type */
    char lname[20];             /* last name */
    char fname[20];             /* first name */
    int age;                    /* age */
    float rate;                 /* e.g. 12.75 per hour */
};

struct tag my_struct;           /* define the structure */
void show_name(struct tag *p);  /* function prototype */

int main(void)
{
    struct tag *st_ptr;         /* a pointer to a structure */
    st_ptr = &my_struct;        /* point the pointer to my_struct */
    strcpy(my_struct.lname,"Jensen");
    strcpy(my_struct.fname,"Ted");
    printf("\n%s ",my_struct.fname);
    printf("%s\n",my_struct.lname);
    my_struct.age = 63;
    show_name(st_ptr);          /* pass the pointer */
    return 0;
}

void show_name(struct tag *p)
{
    printf("\n%s ", p->fname);  /* p points to a structure */
    printf("%s ", p->lname);
    printf("%d\n", p->age);
}

-------------------- end of program 5.2 ----------------
```

Again, this is a lot of information to absorb at one time. The reader should compile and run the various code snippets and using a debugger monitor things like my_struct and p while single stepping through the main and following the code down into the function to see what is happening.

Appendix D

Good-to-Knows

D.1 Matlab

Author: Saroj Saimek original
Brad Kratochvil extended

Date: 1999-2004

Version: 0.2

D.1.1 Background

MATLAB is short for *Matrix Laboratory*. The first version of MATLAB, originally intended to be used only for matrix math, was written during the late 70's at the University of New Mexico and Stanford University. Today MATLAB has become one of the most widely used mathematical software tools in the world, with capabilities extending far beyond the manipulation of vectors and matrices. In this short manual, you will be introduced to this tool and shown how to use it to perform simple calculations that can be used as a part of lab 5. The official MATLAB website is `http://www.mathworks.com`. There also many books available in the library and bookstore to give you further insight into the capabilities of this software package.

D.1.2 Starting Point

Since MATLAB is not freeware, you need to have access to a machine that has MATLAB installed. Every machine in the lab should have access to MATLAB. Once you start up MATLAB, you are in the *Command window*, which is primarily where you will interact with MATLAB library subroutines. In the command window you should see the prompt sign.

```
>>
```

If the window is active, a cursor (most likely blinking) should appear to the right of the prompt.

D.1.2.1 Simple Math

The command window works just like a calculator. Suppose you want to add 3, 4 and 5, you can just type

```
>> 3+4+5 <CR>
ans = 12
```

Alternatively, the above problem can be solved by storing information in MATLAB variables:

```
>> A=3 <CR>
A = 3
>> B=4 <CR>
B = 4
>> C=5 <CR>
C = 5
>> result=A+B+C <CR>
result = 12
```

Students can also use -, $*$, /, or $^\wedge$ to do subtraction, multiplication, division and exponentiation, respectively. The order in which these operations are evaluated in a given expression is given by the usual rules of precedence which can be summarized as follow: *Expressions are evaluated from left to right, with the exponentiation operation having the highest order of precedence, followed by both multiplication and division having equal precedence, followed by both addition and subtraction having equal precedence.* Parentheses can be used to alter this usual ordering, in which case evaluation initiates within the innermost parentheses and proceeds outward.

D.1.3 General Purpose Commands

D.1.3.1 General Information

MATLAB has too many commands for most of us to remember. To help you find commands, you can utilize *online help* capabilities. These capabilities include MATLAB commands in the command window; a separate mouse-driven *Help* window; a browser-based help system, etc. Two basic commands that can be used in the command window are `help` and `lookfor`.

D.1.3.1.1 The help command

The `help` command is the simplest way to get help if you know the topic on which you want help. Typing `help topic` displays help about that topic if it exists, e.g.

```
>> help sqrt <CR>
SQRT Square root.
SQRT(X) is the square root of the elements of X. Complex results are
produced if X is not positive.
see also SQRTM
```

Note that the command `sqrt` is capitalized in the help text. When used, however, `sqrt` is never capitalized.

You also can type `help` which will provide guidance to direct you to the exact topic.

```
>> help <CR>
```

D.1.3.1.2 The lookfor command

The `lookfor` command provides help by searching through all the *first* lines of MATLAB help topics and M-files on the MATLAB search path, and returning a list of those that contain the keyword you specify. This keyword need not be a MATLAB command. For example:

```
>> lookfor complex <CR>
```

Try this command on your computer to see how it works.

D.1.3.2 Manging the MATLAB Workspace

In this section we will introduce five commands, `who`, `whos`, `clear`, `save` and `load`, which you will be using quite often. The data and variables created in the *Command* window reside in *MATLAB workspace*. To see what variable names are in the MATLAB workspace, issue the command `who` or `whos`. `whos` will give more detailed information. Experiment with the commands to see the differnce.

```
>> who <CR>
or
>> whos <CR>
```

Command `clear` deletes variables from the MATLAB workspace. For example, suppose in your workspace you have variables, A, a and B:

```
>> clear A <CR>
>> who <CR>
Your variables are:
a B
```

Use `help` to obtain more information about `clear` if you'd like. The last two commands in this section are `save` and `load`. These commands will save and load files on your computer. Use `help` to find out the many formats and types of data that pertain to these commands.

D.1.3.3 An Advanced Topic

It quickly becomes tedious having to type in the same commands everytime you want to use MATLAB. In this section, we will show how you can run a program directly from a file called a *script}* file or an *M-file*. The term *script* symbolizes the fact that MATLAB simply reads from the *script* found in the file. The term *M-file* recognizes the fact that MATLAB script filenames must end with the extension .m, e.g., example.m. To create a script M-file, choose *New* from the file menu and select M-file. This will bring up a text editor window where you can enter MATLAB commands.

D.1.3.3.1 Script Example

The following shows how to do example 1 as an M-file.

```
%Example 1
Knew=100;
zeta=0.1;
wn=300;
Num=Knew;
Den=[1/wn^2 2*zeta/wn 1 0];
bode(Num,Den,{20,1000});
```

The first line is a comment line in the program. MATLAB will not execute anything following a $%$ sign. MATLAB executes the M-file as if the commands were typed in the command window line by line.

D.1.4 Literature Reference

- Hanselman, D. and Littlefield, B. The student Edition of MATLAB: The language of technical computing, Version 5 (User's guide), Prentice Hall, 1997.
- Friedland, B. Control System Design: An introduction to state-space methods, McGraw-Hill, 1986.

D.2 Binary Number System

Authors: Erik Ostergaard, Alexandre Mesot

Date: 2022

Version: 1.1

The Binary Number Base Systems

Most modern computer systems (including the IBM PC) operate using binary logic. The computer represents values using two voltage levels (usually 0V for logic 0 and either +3.3 V or +5V for logic 1). With two levels we can represent exactly two different values. These could be any two different values, but by convention we use the values zero and one. These two values, coincidentally, correspond to the two digits used by the binary number system.

Since there is a correspondence between the logic levels used by the computer and the two digits used in the binary numbering system, it should come as no surprise that computers employ the binary system. The binary number system works like the decimal number system except the Binary Number System:

- uses base 2
- includes only the digits 0 and 1 (any other digit would make the number an invalid binary number)

The weighted values for each position is determined as follows:

2^7	**2^6**	**2^5**	**2^4**	**2^3**	**2^2**	**2^1**	**2^0**	**2^-1**	**2^-2**
128	**64**	**32**	**16**	**8**	**4**	**2**	**1**	**.5**	**.25**

In the United States among other countries, every three decimal digits is separated with a comma to make larger numbers easier to read. For example, 123,456,789 is much easier to read and comprehend than 123456789. We will adopt a similar convention for binary numbers. To make binary numbers more readable, we will add a space every four digits starting from the least significant digit on the left of the decimal point. For example, the binary value 1010111110110010 will be written 1010 1111 1011 0010.

Number Base Conversion

Binary to Decimal

It is very easy to convert from a binary number to a decimal number. Just like the decimal system, we multiply each digit by its weighted position, and add each of the weighted values together. For example, the binary value 1100 1010 represents:

1∗2^7 + 1∗2^6 + 0∗2^5 + 0∗2^4 + 1∗2^3 + 0∗2^2 + 1∗2^1 + 0∗2^0 =

1 ∗ 128 + 1 ∗ 64 + 0 ∗ 32 + 0 ∗ 16 + 1 ∗ 8 + 0 ∗ 4 + 1 ∗ 2 + 0 ∗ 1 =

128 + 64 + 0 + 0 + 8 + 0 + 2 + 0 =

202

Decimal to Binary

To convert decimal to binary is slightly more difficult. There are two methods, that may be used to convert from decimal to binary, repeated division by 2, and repeated subtraction by the weighted position value.

Repeated Division By 2

For this method, divide the decimal number by 2, if the remainder is 0, on the side write down a 0. If the remainder is 1, write down a 1. This process is continued by dividing the quotient by 2 and dropping the previous remainder until the quotient is 0. When performing the division, the remainders which will represent the binary equivalent of the decimal number are written beginning at the least significant digit (right) and each new digit is written to more significant digit (the left) of the previous digit. Consider the number 2671.

Division	Quotient	Remainder	Binary Number
2671 / 2	1335	1	1
1335 / 2	667	1	11
667 / 2	333	1	111
333 / 2	166	1	1111
166 / 2	83	0	0 1111
83 / 2	41	1	10 1111
41 / 2	20	1	110 1111
20 / 2	10	0	0110 1111
10 / 2	5	0	0 0110 1111
5 / 2	2	1	10 0110 1111
2 / 2	1	0	010 0110 1111
1 / 2	0	1	1010 0110 1111

The Subtraction Method

For this method, start with a weighted position value greater that the number.

- If the number is greater than the weighted position for the digit, write down a 1 and subtract the weighted position value.
- If the number is less than the weighted position for the digit, write down a 0 and subtract 0.

This process is continued until the result is 0. When performing the subtraction, the digits which will represent the binary equivalent of the decimal number are written beginning at the most significant digit (the left) and each new digit is written to the next lesser significant digit (on the right) of the previous digit. Consider the same number, 2671, using a different method.

Weighted Value	Subtraction	Remainder	Binary Number
2^12 = 4096	2671 - 0	2671	0
2^11 = 2048	2671 - 2048	623	0 1
2^10 = 1024	623 - 0	623	0 10
2^9 = 512	623 - 512	111	0 101
2^8 = 256	111 - 0	111	0 1010
2^7 = 128	111 - 0	111	0 1010 0
2^6 = 64	111 - 64	47	0 1010 01
2^5 = 32	47 - 32	15	0 1010 011
2^4 = 16	15 - 0	15	0 1010 0110
2^3 = 8	15 - 8	7	0 1010 0110 1
2^2 = 4	7 - 4	3	0 1010 0110 11
2^1 = 2	3 - 2	1	0 1010 0110 111
2^0 = 1	1 - 1	0	0 1010 0110 1111

Binary Number Formats

We typically write binary numbers as a sequence of bits (bits is short for binary digits). We have defined boundaries for these bits. These boundaries are:

Name	Size (bits)	Example
Bit	1	1
Nibble	4	0101
Byte	8	0000 0101
Word	16	0000 0000 0000 0101
Double Word	32	0000 0000 0000 0000 0000 0000 0000 0101

In any number base, we may add as many leading zeroes as we wish without changing its value. However, we normally add leading zeroes to adjust the binary number to a desired size boundary. For example, we can represent the number five as:

Bit	101
Nibble	0101
Byte	0000 0101
Word	0000 0000 0000 0101

We'll number each bit as follows:

1. The rightmost bit in a binary number is bit position zero.
2. Each bit to the left is given the next successive bit number.

Bit zero is usually referred to as the LSB (least significant bit). The left-most bit is typically called the MSB (most significant bit). We will refer to the intermediate bits by their respective bit numbers.

The Bit

The smallest "unit" of data on a binary computer is a single bit. Since a single bit is capable of representing only two different values (typically zero or one) you may get the impression that there are a very small number of items you can represent with a single bit. Not true! There are an infinite number of items you can represent with a single bit.

With a single bit, you can represent any two distinct items. Examples include zero or one, true or false, on or off, male or female, and right or wrong. However, you are not limited to representing binary data types (that is, those objects which have only two distinct values).

To confuse things even more, different bits can represent different things. For example, one bit might be used to represent the values zero and one, while an adjacent bit might be used to represent the values true and false. How can you tell by looking at the bits? The answer, of course, is that you can't. But this illustrates the whole idea behind computer data structures: data is what you define it to be.

If you use a bit to represent a boolean (true/false) value then that bit (by your definition) represents true or false. For the bit to have any true meaning, you must be consistent. That is, if you're using a bit to represent true or false at one point in your program, you shouldn't use the true/false value stored in that bit to represent red or blue later.

Since most items you will be trying to model require more than two different values, single bit values aren't the most popular data type. However, since everything else consists of groups of bits, bits will play an important role in your programs. Of course, there are several data types that require two distinct values, so it would seem that bits are important by themselves. however, you will soon see that individual bits are difficult to manipulate, so we'll often use other data types to represent boolean values.

The Nibble

A nibble is a collection of bits on a 4-bit boundary. It wouldn't be a particularly interesting data structure except for two items: BCD (binary coded decimal) numbers and hexadecimal (base 16) numbers. It takes four bits to represent a single BCD or hexadecimal digit.

With a nibble, we can represent up to 16 distinct values. In the case of hexadecimal numbers, the values 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, and F are represented with four bits. BCD uses ten different digits (0, 1, 2, 3, 4, 5, 6, 7, 8, 9) and requires four bits. In fact, any sixteen distinct values can be represented with a nibble, but hexadecimal and BCD digits are the primary items we can represent with a single nibble.

b3	b2	b1	b0

The Byte

Without question, the most important data structure used by the 80x86 microprocessor is the byte. This is true since the ASCII code is a 7-bit non-weighted binary code that is used on the byte boundary in most computers. A byte consists of eight bits and is the smallest addressable datum (data item) in the microprocessor.

Main memory and I/O addresses in the PC are all byte addresses. This means that the smallest item that can be individually accessed by an 80x86 program is an 8-bit value. To access anything smaller requires that you read the byte containing the data and mask out the unwanted bits.

The bits in a byte are numbered from bit zero (b0) through seven (b7) as follows:

b7	b6	b5	b4	b3	b2	b1	b0

Bit 0 is the low order bit or least significant bit, bit 7 is the high order bit or most significant bit of the byte. We'll refer to all other bits by their number.

A byte also contains exactly two nibbles. Bits b0 through b3 comprise the low order nibble, and bits b4 through b7 form the high order nibble. Since a byte contains exactly two nibbles, byte values require two hexadecimal digits.

Since a byte contains eight bits, it can represent 2^8, or 256, different values. Generally, we'll use a byte to represent:

1. unsigned numeric values in the range 0 => 255
2. signed numbers in the range -128 => +127
3. ASCII character codes
4. other special data types requiring no more than 256 different values. Many data types have fewer than 256 items so eight bits is usually sufficient.

Since the PC is a byte addressable machine, it turns out to be more efficient to manipulate a whole byte than an individual bit or nibble. For this reason, most programmers use a whole byte to represent data types that require no more than 256 items, even if fewer than eight bits would suffice. For example, we'll often represent the boolean values true and false by 00000001 and 00000000 (respectively).

Probably the most important use for a byte is holding a character code. Characters typed at the keyboard, displayed on the screen, and printed on the printer all have numeric values. To allow it to communicate with the rest of the world, the IBM PC uses a variant of the ASCII character set. There are 128 defined codes in the ASCII character set. IBM uses the remaining 128 possible values for extended character codes including European characters, graphic symbols, Greek letters, and math symbols.

The Word

NOTE:

The boundary for a Word is defined as either 16-bits or the size of the data bus for the processor, and a Double Word is Two Words. Therefore, a Word and a Double Word is not a fixed size but varies from system to system depending on the processor. However, for our discussion, we will define a word as two bytes.

For the 8085 and 8086, a word is a group of 16 bits. We will number the bits in a word starting from bit zero (b0) through fifteen (b15) as follows:

b15	b14	b13	b12	b11	b10	b9	b8	b7	b6	b5	b4	b3	b2	b1	b0

Like the byte, bit 0 is the LSB and bit 15 is the MSB. When referencing the other bits in a word use their bit position number.

Notice that a word contains exactly two bytes. Bits b0 through b7 form the low order byte, bits 8 through 15 form the high order byte. Naturally, a word may be further broken down into four nibbles. Nibble zero is the low order nibble in the word and nibble three is the high order nibble of the word. The other two nibbles are "nibble one" or "nibble two".

With 16 bits, you can represent 2^16 (65,536) different values. These could be the unsigned numeric values in the range of 0 => 65,535, signed numeric values in the range of -32,768 => +32,767, or any other data type with no more than 65,536 values. The three major uses for words are

1. 16-bit integer data values
2. 16-bit memory addresses
3. any number system requiring 16 bits or less

The Double Word

A double word is exactly what its name implies, two words. Therefore, a double word quantity is 32 bits. Naturally, this double word can be divided into a high order word and a low order word, four bytes, or eight nibbles.

Double words can represent all kinds of different data. It may be

1. an unsigned double word in the range of 0 => 4,294,967,295,
2. a signed double word in the range -2,147,483,648 => 2,147,483,647,
3. a 32-bit floating point value
4. any data that requires 32 bits or less.

Assignments Between Types of Different Sizes

In Section C.4, type definitions to specify the length of a variable are given. Types `intN_t` and `uintN_t` are handy, however they may not be available or easy to access on all platforms. This generally does not pose a significant issue, however there are exceptions. For instance, when there is an active communication with a different system, could potentially be an embedded device, and an 8-bit unsigned integer value is expected on the other side. For this kind of situations there are alternative ways to control the number of bits to be used. Here we are only going to mention one of those. In C, the smallest addressable unit is a byte (8-bits). Typically characters are stored in 8 bits with the ASCII (American Standard Code for Information Interchange) value of the associated symbol (see Sec. D.2, Byte). By definition, ASCII values range from 0 to 255, which coincides with the range of an 8-bit unsigned integer. So, when the types `intN_t` and `uintN_t` are not available, and there is a need to use specifically 8-bit values, `char` type can be used instead. When

```
char c = 125;
```

is called, the 8 bit unsigned binary value of 125 is stored in `c`. This can be extended even further. Now, assume you need to use 16 bit values or buffers for some reason.

```
char ca[2];
```

would provide exactly that. Just, this time, you need to think in terms of pointers while using it. Here each character in the array corresponds to a to digit hexadecimal number. When we have

```
ca[0] = 0x67;
ca[1] = 0xab;
```

it has the same effect as having

```
*((unsigned int*)ca) = 0xab67;
```

so the 16-bit value is stored in `ca` in the same way regardless of which assignment is used. Later when we want to access the value we can either pretend that `ca` is a character pointer (`char*`) with 2 characters, or we can pretend that it is an unsigned integer pointer (`unsigned int*`) by explicitly casting it so. As long as we are sure about the memory space that we have allocated, we are fine. In some cases, the compiler might give warnings or even may refuse to complete the compilation. In those cases, try casting the pointer at hand first to a `void*` and then to the type you would like. This approach can easily be extended to non-standard bit counts such as 24. However, be cautious about the type cast you use in those cases, as for instance `unsigned int*` would not be safe. Yet, for lengths like 8, 16, 32, and 64 you can safely use this.

Integer Overflow

Integer overflows occur when the result of an arithmetic operation is a value, that is too large to fit in the available storage space. To make an example:

- Assume you have an unsigned 8-bit variable A = 123.
- A has a range of values from -128 to 127.
- Perform the operation A = A + 10
- This will cause an integer overflow, since 123 +10 = 133, a number which is larger than what can be stored in an uint8 variable.
- The way such an operation is handled is that the variable will be filled up (in our example by 4 up to 127) and the remainder of the now truncated added value will be added to the lower end of the variable range of values (in our example adding the remaining 6 will cause A to flip back to -128 and add up to -123).
- A good way to visualize this is to take an analog counter like in image D.1. When the maximum of the counter is reached (999999) adding 1 will reset it to its lowest value (000000) and any additions will continue from there.

Make sure to keep this in mind when choosing the size and type of your variables.

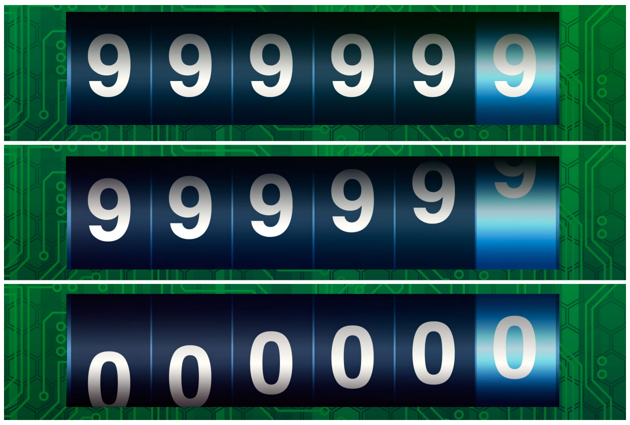


Figure D.1: Analog Integer Overflow