Linux For Embedded Systems

For Arabs

Cairo University
Computer Eng. Dept.
CMP445-Embedded Systems

Ahmed ElArabawy

Lecture 13: Introduction to the Toolchain

Part 1: Build Tools

C Toolchain

- The toolchain is a set of development tools used in association with source code or binaries generated from the source code
- It is associated with development in C/C++ (and other programming languages)
- It is used for a lot of operations such as,
  - Compilation
  - Preparing Libraries
  - Reading a binary file (or part of it)
  - Debugging
- Normally it contains,
  - Compiler : Generate object files from source code files
  - Linker: Link object files together to build a binary file
  - Library Archiver: To group a set of object files into a library file
  - Debugger: To debug the binary file while running
  - And other tools
- Most common toolchain is the **GNU toolchain** which is part of the GNU project
- The GNU toolchain is used whether running with Linux or otherwise

Cross Platform ToolChain

- In PC development, the development machine and the target are the same platform
- This means, the toolchain runs on the same platform that will be used to run the code
- In embedded systems, the toolchain runs on a host platform, which is different from the target platform that will run the binary
- In these cases, we use, **cross platform toolchains**
- For example, we will be using toolchains that run on x86 platforms but it generates binaries (or work on binaries) that would run on a target that will be based on ARM cores
- The developer needs to install the proper toolchain for the used platform
- Most development IDEs (such as **Eclipse**) can be configured with different toolchains depending on the used target

GNU Toolchain

- In this lecture we will be using the GNU toolchain
- There are different releases for the toolchain, for example for the ARM cores,
  - Toolchain for the Raspberry Pi
    https://github.com/raspberrypi/tools
  - Linaro toolchain
    https://launchpad.net/linaro-toolchain-binaries
  - Toolchain for ARM Cortex M and R cores (with no OS or simple RTOS)
    https://launchpad.net/gcc-arm-embedded
- Also, toolchains are included in chip vendors IDE suites such as,
  - CodeSourcery (By Mentor Graphics)
    http://www.mentor.com/embedded-software/sourcery-tools/sourcery-codebench/editions/lite-edition/
  - CodeWarrior (By Freescale)
    http://www.freescale.com/webapp/sps/site/homepage.jsp?code=CW_HOME
  - CodeRed (by NXP)
    http://www.support.code-red-tech.com/CodeRedWiki

Toolchain Components

- The toolchain contains several components
- We will be introducing some of these components in this lecture
- However, each tool in the toolchain is a very rich tool with a lot of capabilities and options, and understanding its details may require much more time
- So this lecture only touches on the surface for these tools and their top uses
- Also, a lot of times the toolchain is used internally by the IDE (such as Eclipse), which means we don't directly use the command line interface for the toolchain
- However, it is still very important to know the different options of each component of the toolchain, since it is highly needed during debugging or for advanced option settings
- The rule is … **NO GUI Can Provide the Power provided by the CLI**

Toolchain Components

Tool	Used for
gcc	Compiling C/C++ (and other languages) code into object files Also it calls a linker internally to link object files into an executable
ld	Links object files into an executable (called internally by gcc)
ar	Archives multiple Object files into a static Library
make	Reads the Makefile to manage the building process
readelf	Reads the contents of an ELF File (object file or executable)
objdump	Reads the internals of an ELF file including assembly code
nm	Reads the symbols inside an object file or and executable
strings	Reads text strings inside a binary file
strip	Strips the binary file from some optional sections
addr2line	Converts an address in the binary to a source file name and line number
size	Display the ELF file section sizes and total size
gdb	Debugger

Toolchain Components

Tool	Used for
gcc	Compiling C/C++ (and other languages) code into object files Also it calls a linker internally to link object files into an executable
ld	Links object files into an executable (called internally by gcc)
ar	Archives multiple Object files into a static Library
make	Reads the Makefile to manage the building process
readelf	Reads the contents of an ELF File (object file or executable)
objdump	Reads the internals of an ELF file including assembly code
nm	Reads the symbols inside an object file or and executable
strings	Reads text strings inside a binary file
strip	Strips the binary file from some optional sections
addr2line	Converts an address in the binary to a source file name and line number
size	Display the ELF file section sizes and total size
gdb	Debugger

Build Tools

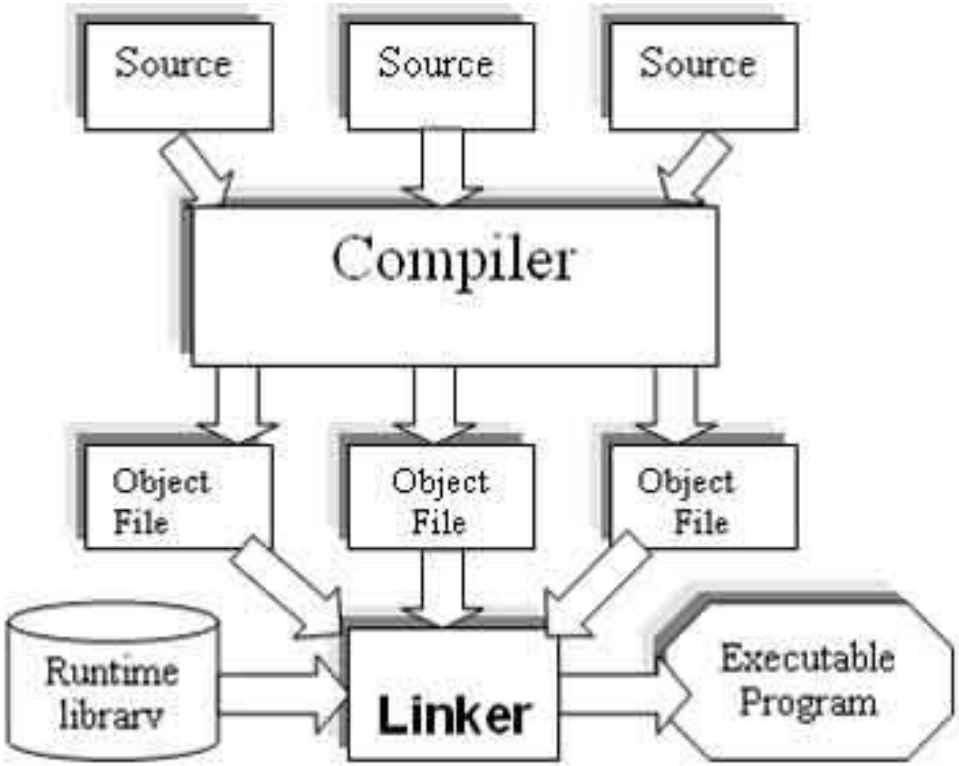
Binary Utilities

GNU Compiler gcc

Compilation Process

- Converting of C source files into an executable happens in 4 Stages:
  - Pre-Processing
    - Expands the #'s (#define, #if, #include, ….)
  - Compilation
    - Convert C source files into assembly code
  - Assembly
    - Convert assembly into an object file
  - Linking
    - Combine the object files (and static libraries) into an executable
- First 3 steps occur separately on individual source files to generate the relocatable object files (*.o)
- This means we will have one object file per source file
- The 4th step works on multiple object files to generate the executable file

GNU gcc Command Compilation Process



GNU gcc Command Basic Usages

$ gcc [options] <files to work on>

- To perform complete compilation process of the main.c source

  $ gcc main.c

  This command will generate the **a.out** executable (assuming the program is composed of a single file)
- To change the name of the generated executable

  $ gcc main.c -o test

  This command will generate an executable with the name test
- To specify the Include path (path for header files)

  $ gcc main.c -I/usr/share/include -I. -I./inc/ -I../../inc
- To enable all warning during compilation process

  $ gcc -Wall main.c -o test
- To convert warnings into errors

  $ gcc -Wall -Werror main.c -o test
- To pass options to gcc in a file (instead of in the command)

  $ gcc main.c @options-file

  Where options-file is a text file that will contain the required options

GNU gcc Command Controlling Optimization Level

- The GCC Compiler performs its job to meet different criteria such as,
  - Small image foot print (small object file size)
  - Efficient memory usage
  - Fast binary processing speed
  - Fast compilation speed
  - Maintain some instrumentation code for debugging purposes
- The developer can set the compiler to optimize for one (or more) of these criteria on the expense of other criteria (for example better memory usage and processing time on the expense of compilation speed and debugging support)
- This is performed using the **-O** option
- There are multiple levels (those are some examples):

Optimization Level	Effect
-O0	Default Level
-O1, -O2, -O3	Optimize for processing speed
-Os	Optimize for image size
-Og	Optimize for debugging capabilities

GNU gcc Command Partial Compilation

- To stop after Pre-Processing

  $ gcc -E main.c > main.i

  This will only generate main.i file which is a pre-processed source file
- To stop after compilation

  $ gcc -S main.c > main.s

  This will generate an assembly file main.s
- To stop after assembly

  $ gcc -c main.c

  This will generate a **relocatable object** file main.o
- To continue all the way, but saving the intermediate files

  $ gcc -save-temps main.c -o test

  This will save:
  - File after pre-processing (**main.i**)
  - File after compilation (**main.s**)
  - File after assembly (**main.o**)
  - Executable (**test**)

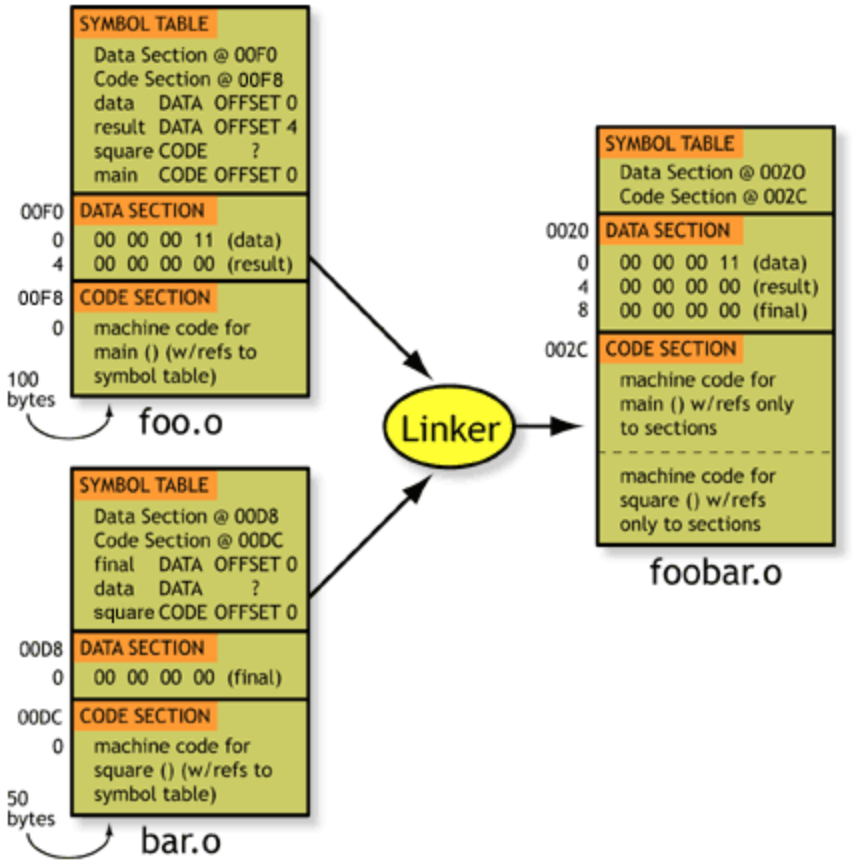
Object Files

- The Pre-processing/Compilation/Assembly steps act on a single source file (independent of other files in the project)
- The outcome is called a **Relocatable Object File**
- The object file generated by the GNU toolchain follows the **ELF file format** (To be discussed later in this lecture)
- Besides the binary code of the program, ELF files contain other info such as a **symbol table** which contain a list of symbols generated during the compilation process along with their addresses in the file
- Symbols in the Symbol table include (among other things):
  - Names of static and global variables defined in the file
  - Names of static and global variables used in the source file
  - Functions defined by the source file
  - Functions used (called) within the source file
- Symbols used in the file can be either,
  - Defined in the same file (defined symbols)
  - Not be defined in the file (undefined symbols)

Linking Process

- The linker job is to connect the dots by combining multiple object files using the symbol table in each one of them
  - If the used symbol is defined in the same file, then replaces the symbol with an address of its location
  - If the used symbol is undefined in the same file, then it looks for the address of that symbol in other object files
    - Object files generated from other source files
    - Library files passed to the Linker
- The outcome of the linking process is an executable that does not rely on symbols, since all of them are replaced by addresses (resolved)
- Hence, the symbol table is essential for the linking process, and is not needed to run the executable
- However, the symbol table can be useful when running the debugger

GNU gcc Command Linking Process

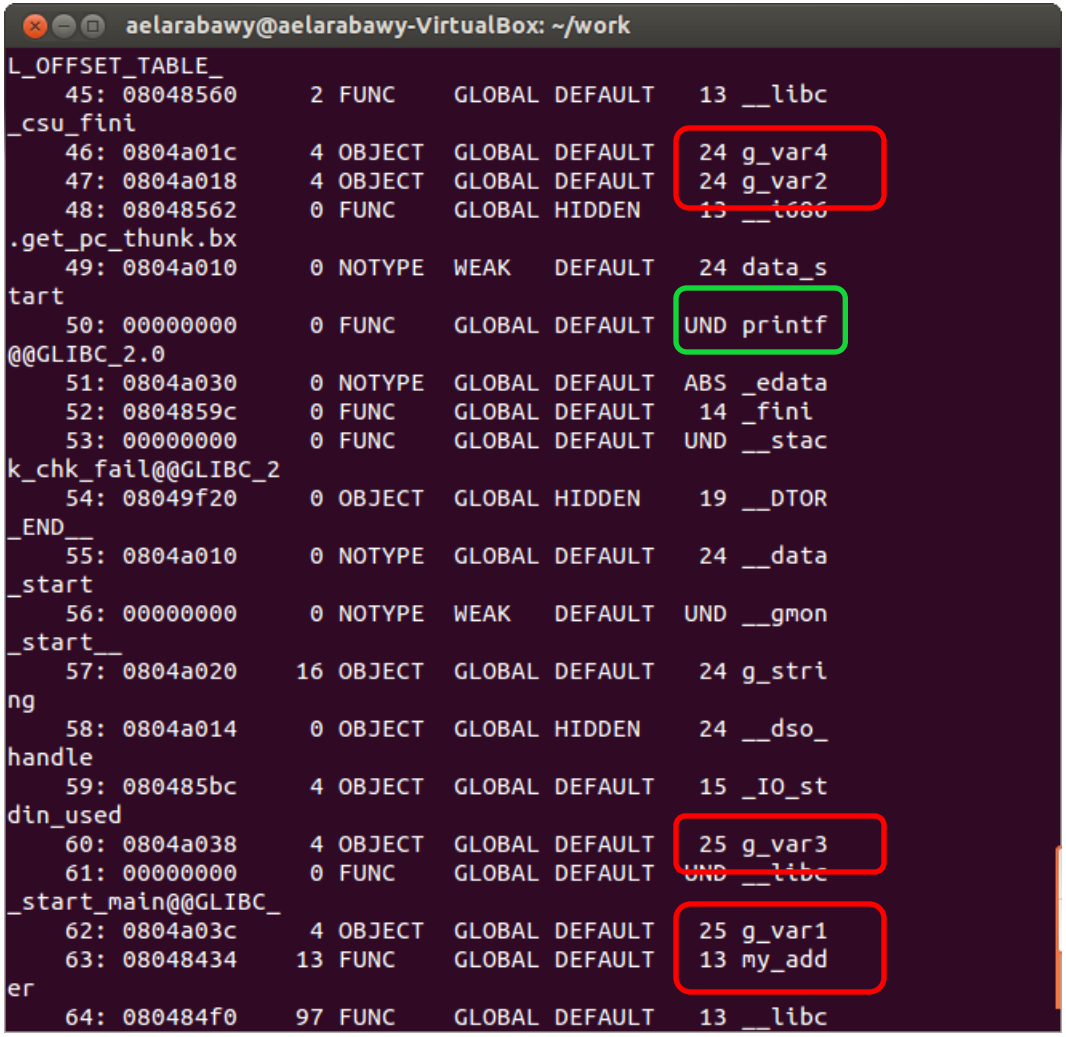


Example: Code

aelarabawy@aelarabawy-VirtualBox: ~/work

```
#include <stdio.h>


int g_var1;
int g_var2 = 5;
int g_var3;
int g_var4 = 6;
char g_string[] = "Global variable";

int my_adder(int x, int y) {
    return x+y ;
}

int main() {
    int i;
    int first_var = 5;
    char my_string[] = "Hello there ";

    for (i = 0; i < first_var; i++ ) {
      printf ("%d / %d / %d  %s %s \n", i, g_var1, g_var2, my_string, g_string);
    }

    return 0;
}
```

Example: Symbol Table



Linking Process (ld Command)

$ ld [options] <object file(s)>

- In GNU, the linking is performed using the ***ld*** tool
- Examples:

  $ ld file1.o file2.o main.o

  $ ld -o my-bin-file file1.o file2.o main.o
- However, no need to call it directly, since ***gcc*** calls it internally

  $ gcc file1.o file2.o main.o

  $ gcc -o my-bin-file file1.o file2.o main.o
- Also, ***gcc*** can perform linking as part of its full handling of source files

  $ gcc -Wall file1.c file2.c main.c -o my-bin-file
- Note: It is recommended to use **gcc** and not **ld** directly, since **gcc** will take care of some details, that the developer will need to handle himself if using **ld** directly

Dealing with Libraries

- Normally the program functionality are located in:
  - The program source code (functionality specific to this program)
  - Some pre-compiled Libraries (functionality that is used by multiple programs, such as printing, writing to files, …)
- Hence the program needs to be **linked** to some libraries in addition to the object files of the source files of the program
- Libraries can be,
  - Static Libraries:
    - A static library is a simple archive of pre-compiled object files
    - Linking of a static library occurs at the same time of object files of the program
    - The library becomes a part of the executable image
  - Dynamic Libraries (shared objects):
    - Linking occurs at run time
    - Hence the executable image does not contain the required functionality
    - The shared object should be available to the executable at run time

Static versus Dynamic Linking

- **So why do we use Dynamic Linking ??**
  - To keep the executable binary image size smaller
  - Since multiple programs may be using the same library, it makes no sense to include it in each and every one of them
  - Also, as the different programs loaded in memory, they need less memory (since the shared object will be loaded only once in the memory)
  - Upgrade in functionality and bug fixing of the library does not require re-building of all the programs using this library

- **Why do we use Static Linking ??**
  - To remove dependencies (the program has everything it needs to run)
  - To make sure we are using a specific version of the library (to avoid conflicts)
  - Used normally with libraries that are not common to be used by other programs

Creating a Static Library (ar Command)

$ ar [options] <library name> <object files>

- The ***ar*** command is used to create or manage static library files
- The static library is a simple archive of pre-compiled object files
- Note that library object files should not contain a main function
- Static library names should should start with "**lib**" and end with "**.a**" such as: libmath.a , libcontrol.a, libvision.a
- To create a static library file

  $ ar rcs libmylib.a file1.o file2.o
- To add another object file to the library

  $ ar r libmylib.a file3.o
- To remove an object file from the library

  $ ar d libmylib.a file3.o
- To view the object files in the library

  $ ar t libmylib.a
- To extract object files from the library

  $ ar x libmylib.a

Linking with Static Libraries

- Static libraries are linked using same commands for linking program object files
- To link the libmath.a library with the program

  $ ld -d n -o my-bin-file file1.o file2.o main.o -lmath

  Note that "**-d n**" forces the linking to be static
- To define the directory to search for the library

  $ ld -d n -o my-bin-file file1.o file2.o main.o -L/home/user -lmath

  $ ld -d n -o my-bin-file file1.o file2.o main.o -L. -lmath
- However, it is recommended to use **gcc** which calls **ld** internally

  $ gcc -static -Wall file1.c file2.c -L. -lmath -o my-bin-file

  $ gcc -static -Wall file1.c file2.c -l:libmath.a -o my-bin-file

Creating a Shared Object

- When we want our code to be compiled into a shared object to be used by other programs (instead of creating an independent executable)
  - At compilation time, we need to make sure our object files are "Position Independent"

    $ gcc -c -Wall –Werror -fpic file1.c file2.c
  - Then we need to create the shared object

    $ gcc -shared -o libmy-shared-object.so file1.o file2.o
- Note that the shared object name must start with "**lib**" and end with "**.so**"

Linking with Shared Object

- Shared objects are linked using same commands used for static libraries (with some option changes)
- Note that in dynamic linking, the shared object image is not copied in the executable image, hence, we will need the shared object at run time as well
- To link the libmath.so library with the program

  $ ld -d y -o my-bin-file file1.o file2.o main.o -lmath

  Note that "**-d y**" forces the linking to be dynamic
- To define the directory to search for the shared object

  $ ld -d y -o my-bin-file file1.o file2.o main.o -L/home/user -lmath

  $ ld -d y -o my-bin-file file1.o file2.o main.o -L. -lmath
- However, normally we use **gcc** which calls **ld** internally

  $ gcc -Wall file1.c file2.c -L. -lmath -o my-bin-file

  $ gcc -Wall file1.c file2.c -l:libmath.so -o my-bin-file

  Note that the default for gcc is to use dynamic linking (unless **-static** is used)

At Run time:

- When the program loads in memory (at run time), the program binary is checked by the **Dynamic Linker** (***ld-linux.so***)
- Note that ld-linux.so is dynamically linked by ***gcc*** with the program by default when using dynamic linking
- The Dynamic linker checks for dependencies on shared libraries
- It also tries to resolve these dependencies by locating the required libraries and updating the binary
- If the Dynamic linker fails to find a library, the program fails to run
- **So, how does dynamic linker finds the required libraries ??**

Dynamic Linker Finds the Libraries,

Via Library Path Environment Variable:

- It checks the environment variable ***LD_LIBRARY_PATH***, which contain a list of directories (separated by a colon "**:**") that contain the libraries

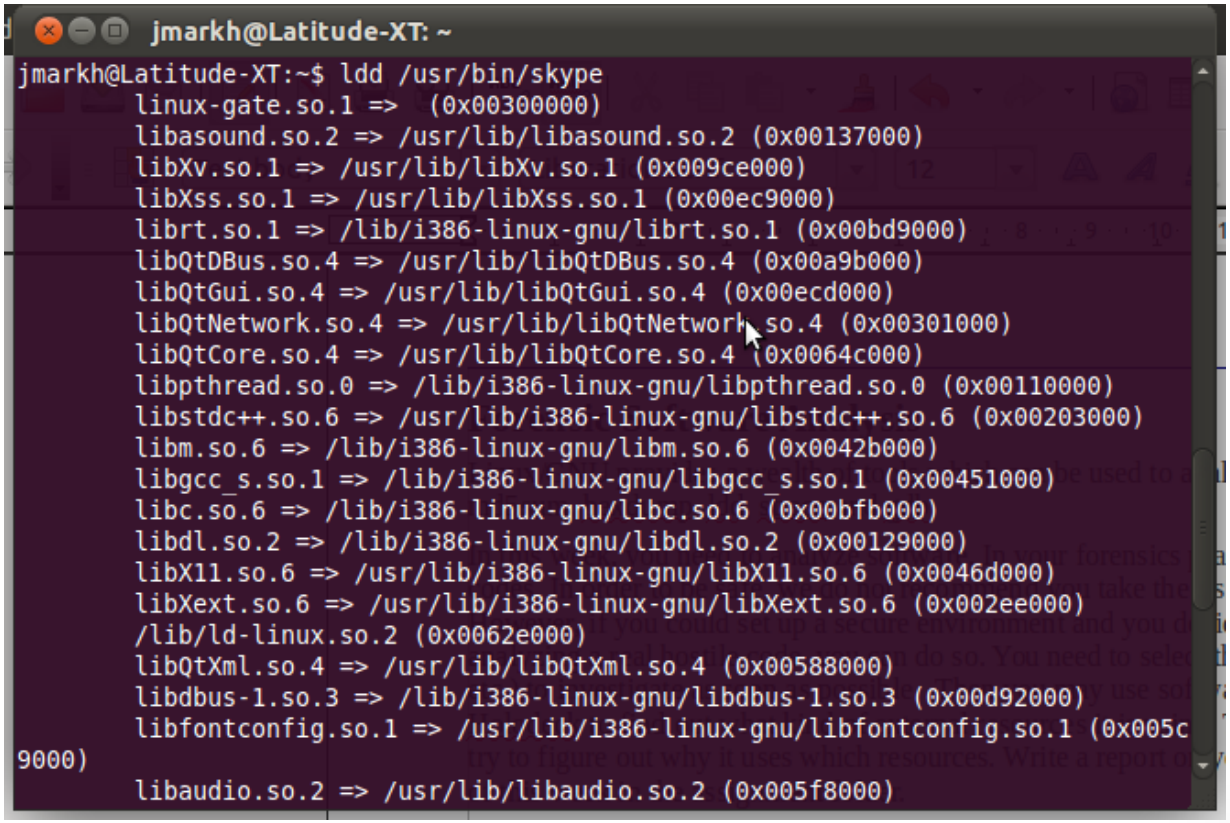
Via Dynamic Linker Cache

- The dynamic linker cache is a binary file (***/etc/ld.so.cache***)
- It contains an index of libraries and their locations
- It provides a fast method for the dynamic linker to find libraries in the specified directories
- To add a directory to the dynamic linker cache,
  - Add the directory path to the file ***/etc/ld.so.conf***
  - Run the utility ***ldconfig*** to generate the binary cache

Finding the Required Libraries (The ldd Command)

$ ldd <program>

- This command lists the required libraries (along with their location) for this program
- Multiple programs can be specified for the command



Managing the Dynamic Linker Cache File (The ldconfig Command)

$ ldconfig

$ ldconfig <Library Directories>

- This command is used to display contents, or build the Dynamic Linker Cache file (***/etc/ld.so.cache***)
- To display the cache contents

  $ ldconfig -p
- To build the cache from the ***/etc/ld.so.conf*** file (in addition to ***/lib*** and ***/usr/lib***)

  $ ldconfig
- To build the cache as above, with the addition of /usr/local/lib,

  $ ldconfig /usr/local/lib

The Building Process

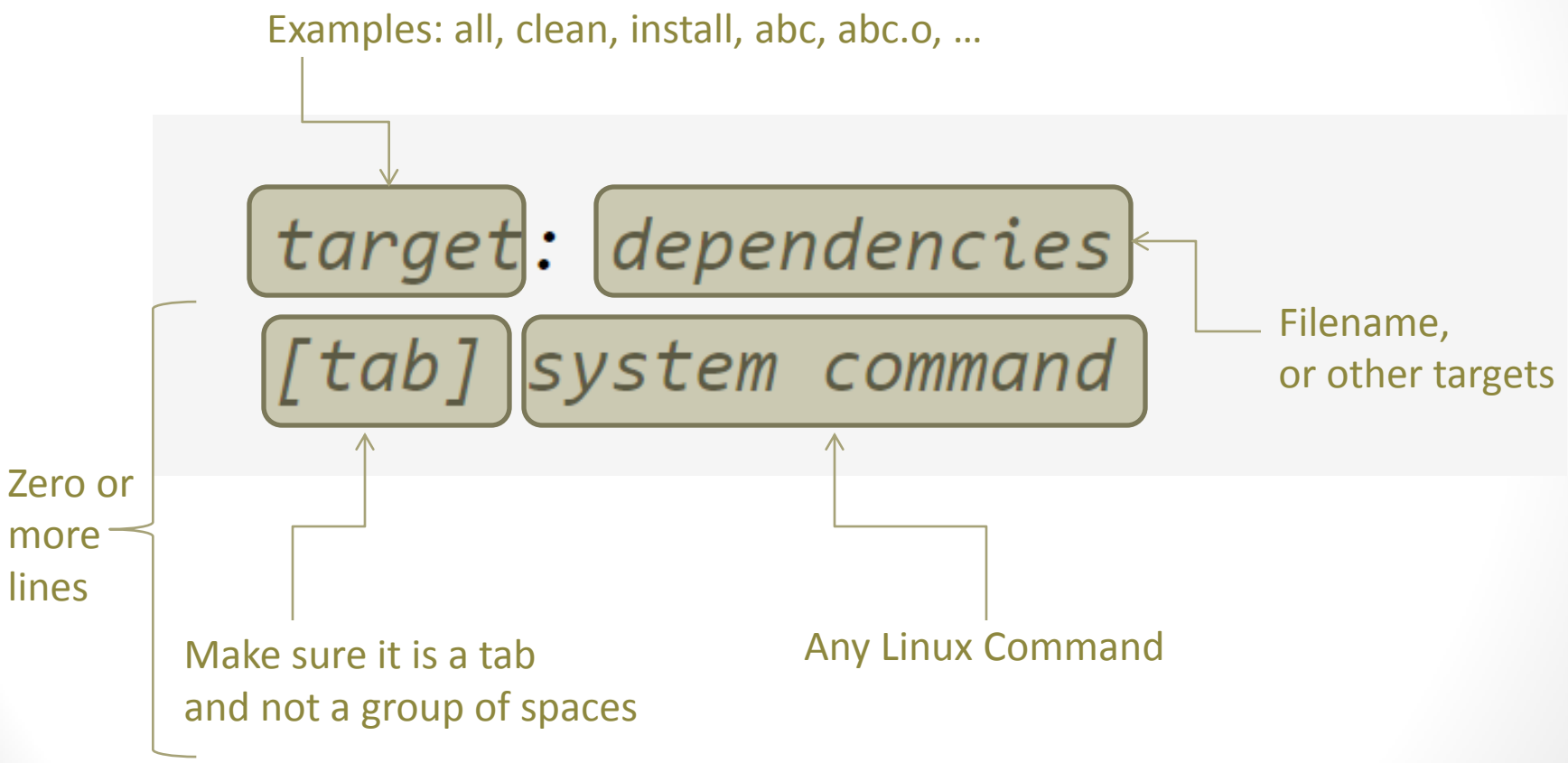
- So far we discussed,
  - Compilation of source code files into relocatable object files
  - Linking object files into an executable
  - Creating static libraries
  - Creating shared objects
  - Linking with static libraries and/or shared objects
- In a normal program, we will be dealing with
  - A lot of source files
  - Some of the source files need to be compiled into static libraries
  - Other source files will need to be compiled into shared objects
  - The executable will need to be linking with some shared objects and some static libraries
    - The ones generated by our code
    - Other libraries and shared objects
- Hence the building process is going to be too complicated, we need a tool to manage all of that for us

GNU make

GNU make

- GNU make is a tool to manage the build process for a software project
- It reads the set of **build rules/targets** inside a **Makefile** and use it to perform the build
- The **Makefile** contains a set of rules/targets, associated with some dependencies
- When a developer issue the make tool, he will need to specify which target to build (if no target is specified, the first target in the file is assumed)
- GNU make will go to that target, and test its pre-conditions,
  - If all the pre-condition rules are satisfied, then it will perform the actions associated with this target
  - If one (or more) pre-conditions is(are) not satisfied, then GNU Make moves to the unsatisfied rules, and perform the same test
- This means we end up with a tree hierarchy of rules, each rule has a set of actions that needs to be performed to satisfy it

A Simple Makefile Rule



A Simple Makefile

```
all: hello

hello: main.o factorial.o hello.o
    g++ main.o factorial.o hello.o -o hello

main.o: main.cpp
    g++ -c main.cpp

factorial.o: factorial.cpp
    g++ -c factorial.cpp

hello.o: hello.cpp
    g++ -c hello.cpp

clean:
    rm *o hello
```

A Simple Makefile

$ make

```
all: hello
```

```
hello: main.o factorial.o hello.o
    g++ main.o factorial.o hello.o -o hello
```

```
main.o: main.cpp
    g++ -c main.cpp
```

```
factorial.o: factorial.cpp
    g++ -c factorial.cpp
```

```
hello.o: hello.cpp
    g++ -c hello.cpp
```

```
clean:
    rm *o hello
```

A Simple Makefile

```
all: hello
```

```
hello: main.o factorial.o hello.o
    g++ main.o factorial.o hello.o -o hello
```

```
main.o: main.cpp
    g++ -c main.cpp
```

```
factorial.o: factorial.cpp
    g++ -c factorial.cpp
```

```
hello.o: hello.cpp
    g++ -c hello.cpp
```

$ make clean

```
clean:
    rm *o hello
```

Another Makefile

```
# I am a comment, and I want to say that the variable CC will be
# the compiler to use.
CC=g++
# Hey!, I am comment number 2. I want to say that CFLAGS will be the
# options I'll pass to the compiler.
CFLAGS=-c -Wall

all: hello

hello: main.o factorial.o hello.o
    $(CC) main.o factorial.o hello.o -o hello

main.o: main.cpp
    $(CC) $(CFLAGS) main.cpp

factorial.o: factorial.cpp
    $(CC) $(CFLAGS) factorial.cpp

hello.o: hello.cpp
    $(CC) $(CFLAGS) hello.cpp

clean:
    rm *o hello
```

Simple Makefile

aelarabawy@aelarabawy-VirtualBox: ~/work/projects/software-projects/dbus/src

```
#########################################
## Makefile
## Author: Ahmed ElArabawy
## Created on : 16 Nov. 2014
#########################################

# ToolChain
RPI_TOOLS_DIR=/usr/bin
CC=$(RPI_TOOLS_DIR)/gcc -std=gnu99

# Binary Name
BIN_NAME=intro-dbus

# Include Directory
INCLUDE_DIR=../include
## Add any more include folders here
CFLAGS=-I$(INCLUDE_DIR) -I/usr/include/glib-2.0 -I/usr/lib/i386-linux-gnu/glib-2.0/include

SRC_DIR=../src
LIB_DIR=../lib
BIN_DIR=../bin

OBJ_DIR=../build
#LIBS= -lgio-2.0
LIBS= -l:libgio-2.0.so.0.4000.0 -l:libglib-2.0.so.0.4000.0 -l:libgobject-2.0.so.0.4000.0
DEP=
```

Simple Makefile

aelarabawy@aelarabawy-VirtualBox: ~/work/projects/software-projects/dbus/src

```
# No Customization below this line
########################################
_OBJ=$(BIN_NAME).o
OBJ=$(patsubst %,$(OBJ_DIR)/%,$(_OBJ))


all: $(BIN_DIR)/$(BIN_NAME)

$(BIN_DIR)/$(BIN_NAME): $(OBJ)
        $(CC) -o $@ $^ $(CFLAGS) $(LIBS)

$(OBJ_DIR)/%.o: %.c $(DEP)
        $(CC) -c -o $@ $< $(CFLAGS)

.PHONY: clean
clean:
        rm -f $(OBJ_DIR)/*.o *~ core $(INCLUDE_DIR)/*~


.PHONY: install
install:
        echo "Nothing to Install"
```

Linux 4 Embedded Systems

http://Linux4EmbeddedSystems.com