

T.R.
GEBZE TECHNICAL UNIVERSITY
GRADUATE SCHOOL OF NATURAL AND APPLIED SCIENCES

**SHARED OBJECT BUILDING MAKEFILES WITH
APPLICATIONS FOR IP PROTECTION**

ÖNDER GÖRMEZ
**A THESIS SUBMITTED FOR THE DEGREE OF
MASTER OF SCIENCE**
DEPARTMENT OF ELECTRONIC ENGINEERING

GEBZE
2020

T.R.
GEBZE TECHNICAL UNIVERSITY
GRADUATE SCHOOL OF NATURAL AND APPLIED SCIENCES

SHARED OBJECT BUILDING
MAKEFILES WITH APPLICATIONS
FOR IP PROTECTION

ÖNDER GÖRMEZ
A THESIS SUBMITTED FOR THE DEGREE OF
MASTER OF SCIENCE
DEPARTMENT OF ELECTRONIC ENGINEERING

THESIS SUPERVISOR
ASSIST. PROF. DR. ÖNDER ŞUVAK

GEBZE
2020

T.C.
GEBZE TEKNİK ÜNİVERSİTESİ
FEN BİLİMLERİ ENSTİTÜSÜ

SİNAİ HAKLARIN KORUNMASI İÇİN
PAYLAŞIMLI NESNE DERLEYEN
MAKEFILE TASARIMI VE
UYGULAMALARI

ÖNDER GÖRMEZ
YÜKSEK LİSANS TEZİ
ELEKTRONİK MÜHENDİSLİĞİ ANABİLİM DALI

DANIŞMANI
DR. ÖĞR. ÜYESİ ÖNDER ŞUVAK

GEBZE
2020

GTÜ Fen Bilimleri Enstitüsü Yönetim Kurulu'nun 23/01/2020 tarih ve 2020/05 sayılı kararıyla oluşturulan jüri tarafından 31/01/2020 tarihinde tez savunma sınavı yapılan Önder GÖRMEZ'in tez çalışması Elektronik Mühendisliği Anabilim Dalında YÜKSEK LİSANS tezi olarak kabul edilmiştir.

JÜRİ

ÜYE

(TEZ DANIŞMANI) : DR. ÖĞR. ÜYESİ ÖNDER ŞUVAK



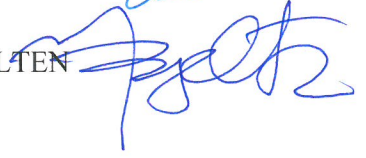
ÜYE

: DOÇ. DR. SERDAR SÜER ERDEM



ÜYE

: DR. ÖĞR. ÜYESİ MUSTAFA BERKE YELTEN



ONAY

Gebze Teknik Üniversitesi Fen Bilimleri Enstitüsü Yönetim Kurulu'nun
...../...../..... tarih ve/..... sayılı kararı.

SUMMARY

Within the scope of this thesis, in order to take advantage of the flexibility provided by 3rd party scientific libraries, to increase the interoperability of programs, to compile and dynamically link C/C++ source codes, to find dependencies by scanning a folder structure of a project, a complex, comprehensive, easy-to-use and automation-based generic makefile project is designed.

Through this makefile, 3rd party libraries will be compiled to create shared objects that can be linked to other programs during runtime.

It is expected that the projects to be compiled by makefile will be hosted in a folder hierarchy. The folder hierarchy is automatically recognized by the makefile, the libraries needed for the projects will be automatically detected and the conflicts of library dependencies will be resolved.

At the end of these processes, the projects are compiled into executable programs. The compiled shared object libraries will be linked to this executable during run time. How the program behaves at run time will be determined by code snippets in these libraries.

Thanks to the generic nature of the makefile project, all the code can be compiled with a single command and application files can be generated without the need to deal with low-level details during the compilation and linking of libraries. In this way, those who want to develop their projects in Linux environment will be isolated from the dependencies of 3rd party libraries and will save time by using these libraries without any configuration and will focus on the technical details of their projects. In this way, the duration of scientific studies is significantly reduced.

Key Words: Makefile, Generic Makefile, Shared Object, Shared Object Library, Linux.

ÖZET

Bu tez çalışması kapsamında, 3. parti bilimsel kütüphanelerin sağladığı esneklikten faydalanmak, programların birlikte çalışabilirliğini arttırmak, gerek C/C++ kaynak kodlarının derlenme ve bağlanma işlemlerinin yapılması, gerekse bir projenin klasör yapısının taranarak bağımlılıkların bulunması ve bu bağımlılıklara göre projenin derleme işlemine geçilmesi için karmaşık, kapsamlı, kullanımı kolay ve otomasyona dayalı bir jenerik makefile tasarımı yapılmıştır.

Bu makefile sayesinde 3. parti kütüphaneler derlenerek çalışma zamanı sırasında başka programlara bağlanabilecek paylaşımlı nesneler oluşturulacaktır.

Makefile tarafından derlenecek projelerin bir klasör hiyerarşisi içerisinde barındırılması beklenmektedir. Klasör hiyerarşisi makefile tarafından otomatik olarak tanınmakta, projeler için gerekli olan kütüphaneler otomatik olarak tespit edilmekte ve kütüphane bağımlılıklarının çakışması ile başa çıkılmaktadır.

Sonrasında projeler derlenerek çalıştırılabilir program haline getirilmektedir. Derlenen paylaşımlı nesne kütüphaneleri run time sırasında bu çalıştırılabilir programa bağlanacaktır. Programın çalışma zamanında nasıl davranacağı bu kütüphanelerdeki kod parçacıkları vasıtasıyla belirlenecektir.

Makefile projesinin jenerik olması sayesinde, araştırmalar yapılırken, kütüphanelerin derlenmesi ve bağlanması aşamasındaki aşağı seviyeli ayrıntılarla uğraşmadan bütün kodlar tek bir komutla derlenebilmekte ve uygulama dosyaları üretilebilmektedir. Böylelikle projelerini Linux ortamında geliştirmek isteyenler 3. parti kütüphanelerin bağımlılıklarından soyutlanarak herhangi bir konfigürasyon yapmadan bu kütüphaneleri kullanarak zaman kazanacak ve projelerinin kendilerini ilgilendiren teknik detaylarına odaklanacaklardır. Bu sayede bilimsel çalışmaların süresi önemli ölçüde azaltılmaktadır.

Anahtar Kelimeler: Makefile, Jenerik Makefile, Paylaşımlı Nesne, Paylaşımlı Nesne Kütüphanesi, Linux.

ACKNOWLEDGEMENTS

I would like to express my special thanks of gratitude to my thesis advisor Asst. Prof. Önder ŞUVAK, who gave me the golden opportunity to do this wonderful project on the topic Shared Object Building Makefiles. He shared his valuable knowledge and experience from the beginning to the end of my graduate education. He not only shared his profound scientific knowledge with me but also taught me great lessons in life. His support, suggestions, and encouragement gave me the drive and will to complete this work. I came to know about so many new things I am really thankful to him.

I would like to thank everyone and especially my friends for helping me to overcome the difficulties I faced during my studies at Gebze Technical University.

Finally, I would like to thank my fiancée Hatice Sıla Balbınar, my father Hasan Görmez, my mother Emiř Görmez, my elder sisters İlknur Görmez, Birsal Demiral and Nermin Çam for valuable supports in this challenging process. I am grateful to my family for their love and support.

TABLE of CONTENTS

	<u>Page</u>
SUMMARY	v
ÖZET	vi
ACKNOWLEDGEMENTS	vii
TABLE of CONTENTS	viii
LIST of ABBREVIATIONS and ACRONYMS	x
LIST of FIGURES	xi
LIST of TABLES	xiii
1. INTRODUCTION	1
2. MAKEFILE	3
2.1. Rules Of Makefile	4
2.2. Macros Of Makefile	5
2.3. Makefile Example	6
3. LIBRARY	10
3.1. Library Naming	11
4. STATIC LINKING	13
4.1. Static Library Usage	13
4.2. Static Library Generation	14
5. DYNAMIC LINKING	16
5.1. Dynamic Linking Example	16
5.1.1. Creating Object File For Using In The Shared Object	16
5.1.2. Generation Shared Object Library	18
5.1.3. Compiling Test Program	18
5.1.4. Unknown Library Path Problem Solutions	20
5.1.4.1. Rpath Option	21
5.1.4.2. Changing LD_LIBRARY_PATH Value	21
5.1.4.3. Using ldconfig	25
6. DYNAMIC LOADING	27
6.1. Dynamic Loading Use Cases	27
6.2. Declerations of Dynamic Loading Functions	27

6.2.1. dlopen Function	28
6.2.2. dlsym Function	28
6.2.3. dlclose Function	29
6.2.4. dlerror Function	30
6.3. Dynamic Loading Example	31
6.3.1. Used Equation in Example Program	31
6.3.2. Generation and Plotting the Data	32
6.3.3. Error Handling	34
7. SHARED OBJECT BUILDING MAKEFILE	36
7.1. Library Makefile	36
7.1.1. Library Folder Detection	37
7.1.2. Library Version Folder Detection	39
7.1.3. Rules	44
7.2. Library Makefile Demo Project	46
7.3. Project Makefile	49
7.3.1. Project Folder Detection	50
7.3.2. Rules	53
7.4. Project Makefile Demo Project	54
8. CONCLUSION AND FUTURE WORK	57
8.1. Conclusion	57
8.2. Future Work	58
REFERENCES	59
BIOGRAPHY	61
APPENDICES	62

LIST of ABBREVIATIONS and ACRONYMS

<u>Abbreviations</u>	<u>Explanations</u>
<u>and Acronyms</u>	
ω	: Omega
API	: Application Programming Interface
BACnet	: Building Automation and Control Networks
GTÜ	: Gebze Teknik Üniversitesi
Hz	: Hertz
IP Protection	: Intellectual Property Protection
PIC	: Position Independent Code
POSIX	: Portable Operating System Interface
RMS	: Root Mean Square
s	: Second
t	: Time
T.C	: Türkiye Cumhuriyeti
T.R	: Republic of Turkey

LIST of FIGURES

<u>Figure No:</u>	<u>Page</u>
2.1: Makefile rule.	4
2.2: Default target built.	7
2.3: Program output.	7
2.4: Hello.elf target built.	8
2.5: Install target built.	8
2.6: Clean target built.	9
3.1: Library files naming convention.	11
4.1: "CalculateArea" library delivery.	13
4.2: Compilation flags for "CalculateArea" library.	14
4.3: Static library generation.	15
5.1: "my_strcmp" and "my_strlen" function output files generation.	18
5.2: "MyStringOp" shared object library generation.	18
5.3: Compilation fails.	19
5.4: Compilation success.	20
5.5: List dynamic dependencies.	20
5.6: Dynamic loading error.	20
5.7: Compiling with "-rpath" option.	21
5.8: Non-exported environment variable.	22
5.9: Exported environment variable.	23
5.10: Running the executable on subshell (wrapper script).	24
5.11: Testing in the command line.	24
5.12: Putting in the standard library location.	25
5.13: Adding "libMyStringOp.conf" file.	26
5.14: Putting in the non-standard library location.	26
6.1: Voltage value at the specified time.	32
6.2: Generated data.	33
6.3: Time-dependent change of mains supply voltage.	34
6.4: Incorrect "dlopen()" call.	34
6.5: Incorrect "dlsym()" call.	35
6.6: Incorrect "dlsym()" call.	35

6.7:	Incorrect "dlclose()" call.	35
6.8:	Incorrect "dlclose()" call.	35
7.1:	Generic library makefile.	36
7.2:	Library folder hierarchy.	37
7.3:	Invalid library folder detection.	38
7.4:	Valid library folder detection.	39
7.5:	"v" prefix control output.	40
7.6:	Version number empty control output.	40
7.7:	"." character presence control output.	41
7.8:	Major version control output.	42
7.9:	Minor version control output.	43
7.10:	Valid library folder detection.	44
7.11:	Rule library output.	45
7.12:	Rule clean output.	46
7.13:	Demo project folder structure.	46
7.14:	Demo project compilation.	48
7.15:	Demo project output files.	48
7.16:	Generic project makefile.	49
7.17:	Project folder hierarchy.	50
7.18:	Invalid project folder detection.	51
7.19:	Invalid project folder detection.	52
7.20:	Valid project folder detection.	53
7.21:	Rule project output.	53
7.22:	Rule clean output.	54
7.23:	Demo Project Folder Structure	54
7.24:	Demo project compilation.	55
7.25:	Demo project output.	56

LIST of TABLES

<u>Table No:</u>	<u>Page</u>
2.1: Make usage.	3
2.2: Example macro definitions.	5
2.3: Makefile test program.	6
2.4: Simple makefile code.	6
3.1: Library file usage in a makefile.	12
4.1: Rectangle area calculation program.	13
4.2: "CalculateArea" library source code.	14
5.1: "my_strcmp" function header and source file.	17
5.2: "my_strlen" function header and source file.	17
5.3: Shared object library test program.	19
6.1: dlfcn.h file.	27
6.2: "dlopen()" and "dlerror()" function usage.	28
6.3: dlsym and dlerror function usage.	29
6.4: "dlclose()" and "dlerror()" function usage.	30
6.5: FPSquareRoot and FPSine function pointers usage.	32
7.1: Library folder detection code.	38
7.2: "v" prefix control code.	39
7.3: Version number empty control code.	40
7.4: "." character presence control code.	41
7.5: Major version control code.	42
7.6: Minor version control code.	43
7.7: Rule library.	44
7.8: Rule clean.	45
7.9: RandomArray.hpp file.	46
7.10: RandomArray.cpp file.	47
7.11: "prj" prefix detection code.	51
7.12: Project name detection code.	52
7.13: Rule project.	53
7.14: Rule clean.	54
7.15: main.cpp file.	55

1. INTRODUCTION

The widespread use of software tools and the development of computer-aided designs have accelerate technological developments. With the widespread use of open-source projects to share information, it has made the work easier of those who are interested in software development and who are testing their academic researches in a computer environment. Providing easier access to open-source software libraries that can be used to solve programmer and researchers problems has made it possible for them to complete their projects quickly without having to write more detailed programs.

While open-source scientific libraries (such as OpenCV, IT++, FFTW) are used, each person has to make the necessary adjustments (create a comprehensive makefile) to use these libraries in their own projects. This process is not a very long process compared to rewriting libraries, but it takes considerable time.

Thanks to the generic nature of the Makefile project, all the code can be compiled with a single command and application files can be generated without the need to deal with low-level details of makefile during the compilation and linking of libraries. In this way, those who want to develop their projects in Linux environment will be isolated from the dependencies of 3rd party libraries and will save time by using these libraries without any configuration and will focus on the technical details of their projects. In this way, the duration of scientific studies is significantly reduced.

This thesis describes the design details and operation of the automation-based makefile project, which will be used by other graduate students working with Mr. Şuvak.

This makefile;

- Can recognize a library folder that located in any location on the computer, compile and link this library with other shared libraries that the library need,
- Shared objects version number can be created automatically according to the version number of the library folder,
- Can solve dependencies on source code and header files grouped as folders during the compilation and can compile and link the files according to these dependencies,

- It can automatically check whether the codes that are included in the project have been changed or not, during recompilation, only the source codes that depend on the changed files can be included in the compilation process,
- The library files (.so files) created after the compilation and linking operations and the header files, which is the interface of these library files, can be automatically collected under a folder that can be used by other projects.

Manually performing these automatic operations performed by makefile, causes serious time loss. In this thesis, details of a makefile design that enable the processes to be performed automatically described above are presented.

This thesis is divided into eight chapters: Chapter 2 describes what the makefile is and what parts it consists of. Chapter 3 describes what library means in programming languages and how to create and use libraries in C/C++. Chapter 4 describes how to create a static library. Chapter 5 describes how to create a dynamic library. Chapter 6 describes how a dynamic library can be linked to a program at run time. Chapter 7 describes the details of the shared object building makefile design, which is necessary to automatically compile and link the libraries mentioned in the preceding chapters. Chapter 8 which is the conclusion chapter is dedicated to the summary of this study, explanation of important points and things that are possible to be done in the future.

2. MAKEFILE

Make is a build automation tool used in software development processes. This program reads files called makefile and automatically compiles the source code and creates executable programs or libraries. Integrated Development Environments (IDEs) and specialized compiler to the language can be used instead of Make. However, the use of Make is very common in Linux-like operating systems due to the lack of these tools, insufficient features or paid.

Make program is created by Stuart Feldman in April 1976 at Bell Lab. The project has grown day by day and has become more comprehensive. For example, GNU Make 4.2 consists of approximately 42,500 lines of code. Make has been used in many large-scale projects such as Linux kernel, Apache OpenOffice and LibreOffice, Mozilla FireFox. To compile GCC, the use of the Make program is compulsory after GCC version 3.4 [1].

The make program searches for makefiles in the current directory. For example, the GNU Make program in Linux checks if the "GNUmakefile", "makefile", or "Makefile" file exists in the working directory. If GNU make program finds one of these files, runs target (or default target) specified in the argument list.

In general, Make is used to detect a change in a source file and to create a target file that covers that change. If any of the targets specified in the makefile are up-to-date, they are not built. The only exceptions are targets with the .PHONY attribute. With this attribute, the Make program is notified that this target is performing a number of preparatory operations that do not contain a file. Therefore, targets that are qualified with this attribute are always rebuilt [2].

Make program is run as follows.

Table 2.1: Make usage.

<code>make [TARGET ARGUMENT2 ...]</code>
--

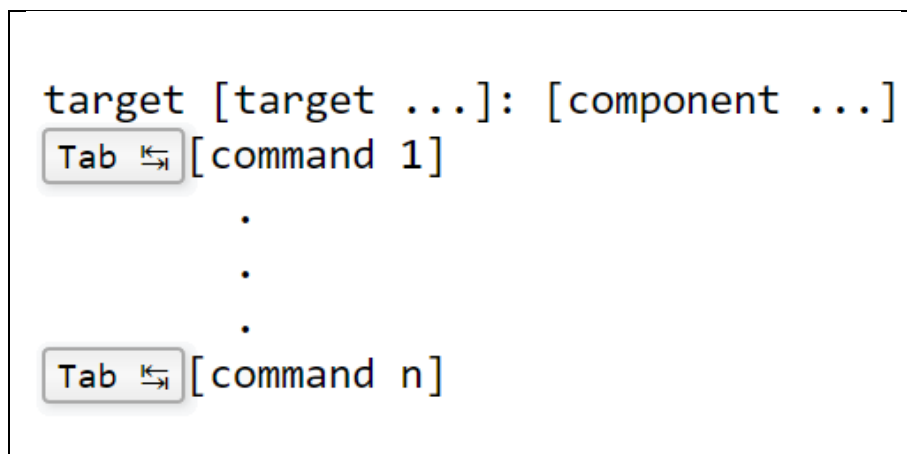
The arguments list is optional. If the argument list is not given when running Make, the first target (default target) in the makefile is built.

Make decides whether to rebuild the target by looking at the time the source files were modified. In this way, output files are not created from targets that do not need to be rebuilt.

2.1. Rules Of Makefile

Typically, makefiles consist of multiple rules. Each rule starts with a name. This name represents the target given when calling the makefile. The naming is terminated by the ":" character. At the same line contains the list of components (other targets or files) to which the target depends. This indicates that components to the right-hand side of the ":" character are prerequisites for the target to the left-hand side of the ":" [2].

The following Figure 2.1 shows a typical rule [1].

The diagram illustrates the syntax of a Makefile rule. It shows a line starting with 'target' followed by '[target ...]' and then ': [component ...]'. Below this, a 'Tab' character is shown with an arrow pointing to the start of the first command line '[command 1]'. This is followed by three dots indicating multiple lines of commands, and then another 'Tab' character with an arrow pointing to the start of the final command line '[command n]'.

```
target [target ...]: [component ...]  
Tab [command 1]  
.  
.  
.  
Tab [command n]
```

Figure 2.1: Makefile rule.

Applying rules to a single target is a more common practice than applying to multiple targets.

The lines following the component line contain commands that tell how to convert components (normally source files) to the target (normally the output). Each command line must be indented using the TAB character. Commands are executed if any of the prerequisites have changed after the last generation time of the target. The commands associated with the rules are called "recipes" in the GNU Make documentation.

Each command line in the recipe is run in a separate shell or command line interpreter. The commands used in the command lines may differ from the operating

system to the operating system. For example, the makefile written for a Linux-like operating system cannot be run on the Windows operating system.

The command lines can be started with one of the three prefixes given below [1].

- Minus or hyphen "-": This character informs the Make program that errors that occur during the execution of the command will be ignored.
- At sign "@": This character informs the Make program that the command should not be printed to the standard output before it is executed.
- Plus sign "+": This character allows the command to be executed even when the Make program is in "do not execute" mode.

2.2. Macros Of Makefile

Macros are variables used to provide many flexibilities to users of the makefile. For example, the "CPP" macro is a commonly used macro instead of the C++ compiler name. So when the user wants to change the C++ compiler, it will only change the macro definition instead of replacing all the usage of compiler name located anywhere in the code ("CPP=clang" instead of "CPP=g++").

Macros can have their initial values with arguments passed during the Make program execution. Environment variables can be assigned to macros. In addition, the output of programs running in the shell can be assigned to macros.

The following Table 2.2 shows typical macros.

Table 2.2: Example macro definitions.

```
LIB_NAME      = $(notdir $(shell dirname "$(CURDIR)"))
PROJECT_NAME  = $(subst $(LIB_TAG),$(empty),$(LIB_NAME))

# Return space added value. For example "libPoissonPointProcess" to "lib
PoissonPointProcess".
# Or return original value "Myproject" to "Myproject"
SPACE_ADDED_LIB_NAME = $(subst $(LIB_TAG),$(LIB_TAG)$(space),$(LIB_NAME))
```

2.3. Makefile Example

A test program was written to give an example of makefile usage. The program, which is given in Table 2.3, prints "Hello world!" words to the standard output stream.

Table 2.3: Makefile test program.

```
#include <iostream>
#include <cstdlib>

using namespace std;

int main(int argc, char **argv)
{
    cout << "Hello world!" << endl;

    exit(EXIT_SUCCESS);
}
```

The following Table 2.4 shows a simple makefile code.

Table 2.4: Simple makefile code.

```
#
# Project      : SharedObjectBuildings
# Author       : Önder Görmez
# Dated        : October 22, 2019
# Description  : Makefile for the simple hello world program.
#

OUTPUT_FILE = hello.elf
INSTALL_DIR = bin
COMPILER = g++

# Default target
.PHONY: all
all: $(OUTPUT_FILE)

# Build hello.elf from main.cpp
$(OUTPUT_FILE): main.cpp
    $(COMPILER) -o $(OUTPUT_FILE) main.cpp

# Copy hello.elf to the bin directory
.PHONY: install
install:
    mkdir -p $(INSTALL_DIR)
    cp -p $(OUTPUT_FILE) $(INSTALL_DIR)

# Delete hello.elf
.PHONY: clean
clean:
    rm -f $(OUTPUT_FILE)
```

There is four target in this makefile. Three of them are .PHONY targets. In other words, if the Make program is run with one of these three targets, the commands given in the recipe section are always executed, regardless of whether the prerequisites are up to date or not. If no target is specified while executing the Make program, "all", which is the first target in the file, is executed. Since the "hello.elf" target is defined as a prerequisite in the "all" target, the status of this target is checked. Since the "main.cpp" source file is defined as a prerequisite in "hello.elf", it is checked whether the source has changed since the last build. If changed, the commands given in the recipe are executed. If not, no action is taken. Figure 2.2 below illustrates this situation.

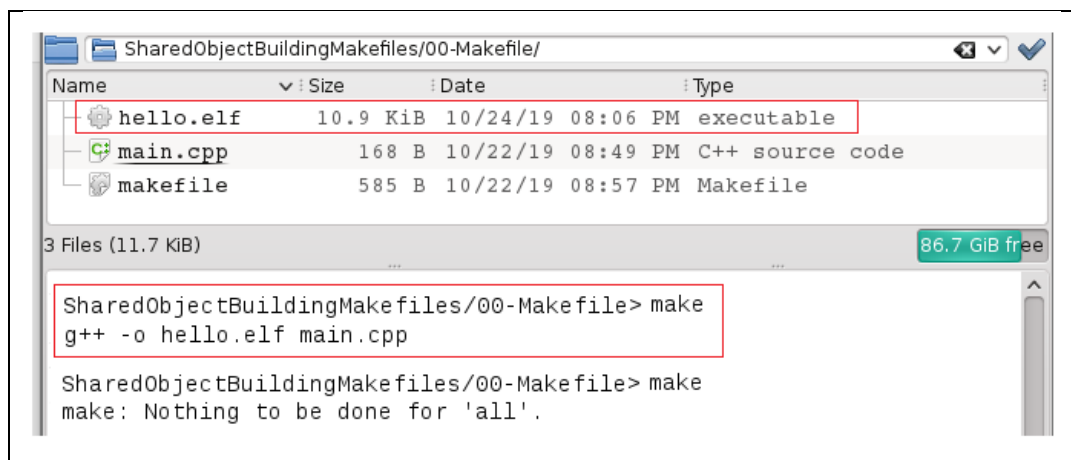


Figure 2.2: Default target built.

The output of the compiled program is as given in Figure 2.3 below.

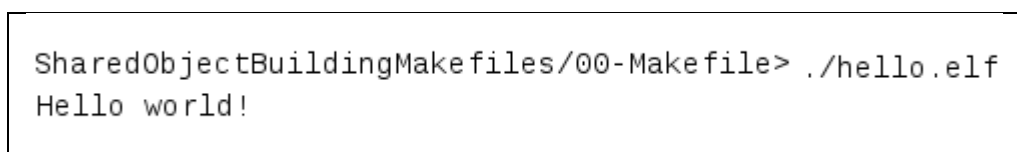


Figure 2.3: Program output.

If the target "hello.elf" is passed to the make program as an argument, an output as in Figure 2.4 below is obtained.

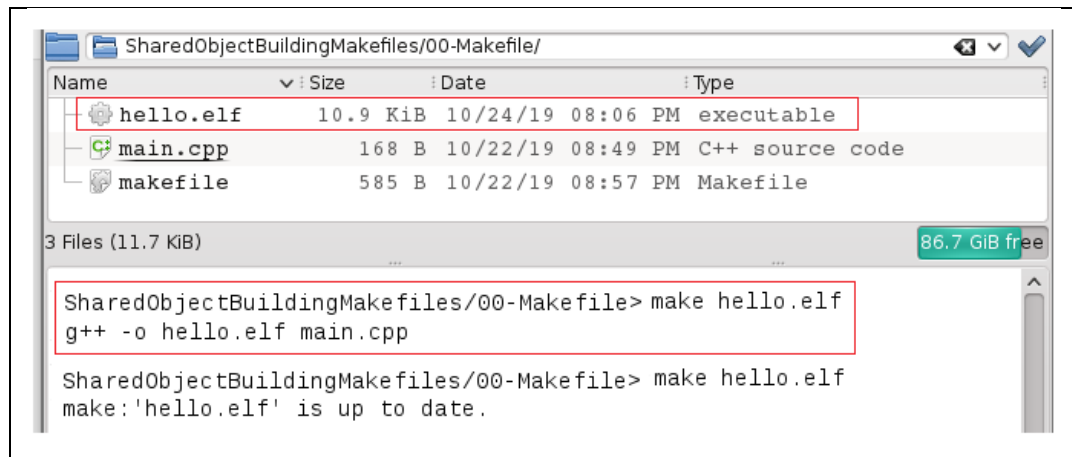


Figure 2.4: Hello.elf target built.

If the target "install" is passed to the make program as an argument, an output as in Figure 2.5 below is obtained.

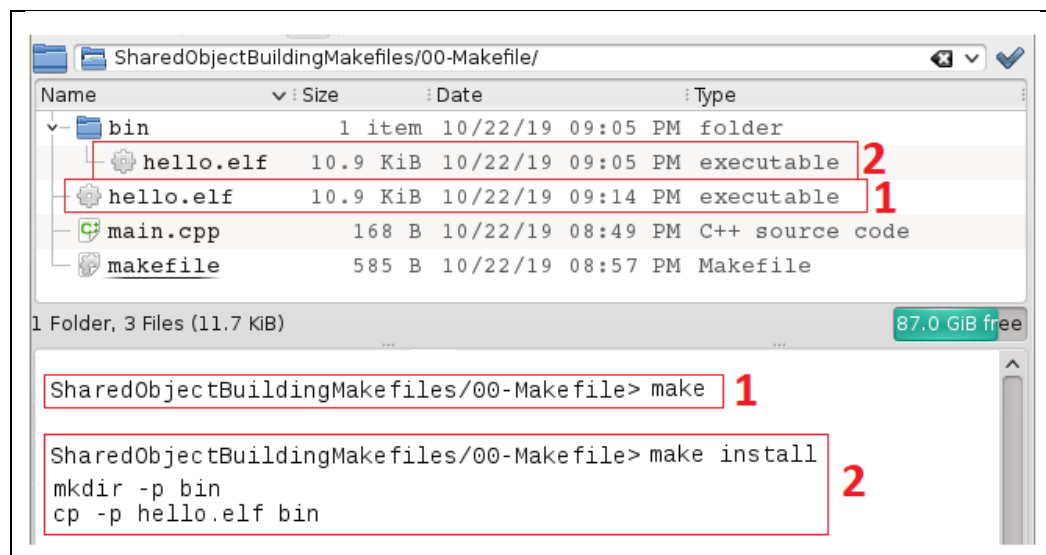


Figure 2.5: Install target built.

If the target "clean" is passed to the make program as an argument, an output as in Figure 2.6 below is obtained.

```
SharedObjectBuildingMakefiles/00-Makefile> ls
bin hello.elf main.cpp makefile

SharedObjectBuildingMakefiles/00-Makefile> make clean
rm -f hello.elf

SharedObjectBuildingMakefiles/00-Makefile> ls
bin main.cpp makefile
```

Figure 2.6: Clean target built.

3. LIBRARY

Libraries are known as a file that contains multiple compiled object code files into a single file. Typically, functions written in the C programming language and classes and methods written in the C++ programming language can be shared by multiple applications. Library files are created by broken out the shared code contained in the applications from the source code of the applications and compiling and bundling them as a different file that can be shared by applications. The standard libraries in C and C++ languages are examples of libraries which can be compiled and linked with your code.

The benefit of libraries arises when very large software components are shared by different applications. For example, suppose 10 applications use 1GB of software components. When these applications include this software component in their source code, 10GB of disk space will be used. Instead, if the 1GB software component is compiled as a separate library file, disk usage will be reduced by 90% as all applications will use the same library file.

Another benefit of using the library is that version management of the library is easy. Multiple versions of libraries can work on the same operating system for different applications.

In addition to all the information given above, libraries provide a strict separation of components from one another. In this way, developers working on the application focus only on the work they need to do. This makes it easier to test and verify the operation of applications [3].

Libraries are basically divided into static libraries and dynamic libraries.

- Static libraries have ".a" file extension. Static libraries are embedded in the object code of the application at the time of linking and become part of the application.
- Dynamic libraries have ".so" file extension. Unlike static libraries, they are not embedded in the object code of the application at the time of linking. This type of library can be used in two different ways.
 - i) It can be dynamically linked to the application for use at run time. In this case, the library must be present during compile and link time. The library

file is not embedded in the application but is tied for use during execution. Therefore, if the library is not found during execution, the application crashes.

- ii) It can be dynamically loaded/unloaded to the application for use at run time. In this case, the library need not present during compile and link time. The library file is not embedded in the application and is not tied during compile and link time for using the execution of the application. If the library is not found during execution, the application continues normal operation. This is achieved by using the dynamic loader in the operating system with system calls [3].

3.1. Library Naming

Libraries that are created must be named with "lib" prefix like any sub-library in the standard C library. The following Figure 3.1 shows the naming convention of preinstalled "dl", "m" and "pthread_nonshared" libraries.

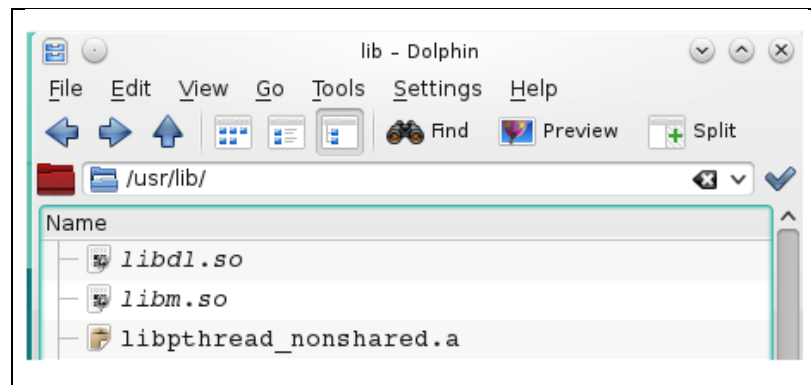


Figure 3.1: Library files naming convention.

"lib" prefix is not used when referring to these libraries during the compilation and linking phase. When referencing the dynamic loading library ("dl"), math library ("m") and POSIX threading library ("pthread_nonshared") located under the "/usr/lib" folder path, the syntax is used as shown in Table 3.1.

Table 3.1: Library file usage in a makefile.

```
CPP          := g++

# Add "-D NDEBUG" flag to CPP_FLAGS, for release optimization
CPP_FLAGS := -Wall -std=c++11 -w

SRCS_CPP    = $(SRC_DIR)/main.cpp
OBSJS_CPP   = $(ROOT)/main.o

main:
    $(CPP) $(CPP_FLAGS) $(SRCS_CPP) -ldl -lm -lpthread_nonshared -o
    ./prog.elf
```

4. STATIC LINKING

Static libraries are a collection of object codes generated from C/C++ source files. They help library developers to hide source code while allowing the software developers to take advantage of library-implemented code while developing their applications.

4.1. Static Library Usage

Typically, libraries are delivered with the library file "libLibraryName.a" and the header file "LibraryName.h", which is the interface to the library. The following Figure 4.1 is an example of the delivery of a library that calculates the area of the rectangle.

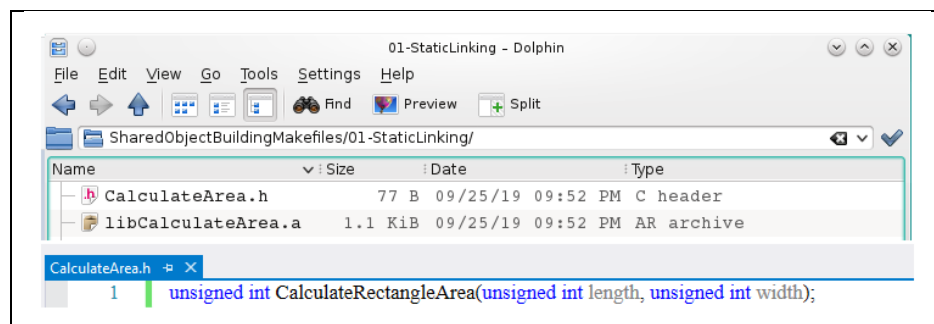


Figure 4.1: "CalculateArea" library delivery.

Using this library, a program such as Table 4.1 can be written to calculate the rectangular area.

Table 4.1: Rectangle area calculation program.

```
#include <iostream>                                     /* NOTE: std::cout */
#include "CalculateArea.h"

using namespace std;

int main(int argc, char **argv)
{
    unsigned int length = 5;
    unsigned int width = 10;

    cout << "Rectangle Length: " << length << endl;
    cout << "Rectangle Width: " << width << endl;
}
```

Table 4.1: Rectangle area calculation program.

```
cout << "Area of Rectangle: " << CalculateRectangleArea(length, width);  
cout << endl << endl;  
  
exit(EXIT_SUCCESS);  
}
```

The program given above can be compiled by adding flags given in red in Figure 4.2. The flags "-L." declares that "CalculateArea" library is on the same path as the program to be compiled.

```
g++ -Wall -std=c++11 main.cpp -L. -lCalculateArea -o prog.elf  
./prog.elf  
Rectangle Length: 5  
Rectangle Width: 10  
Area of Rectangle: 50
```

Figure 4.2: Compilation flags for "CalculateArea" library.

4.2. Static Library Generation

The implementation of the above-mentioned library is as in Table 4.2.

Table 4.2: "CalculateArea" library source code.

```
/*                                CalculateArea.h                                */  
unsigned int CalculateRectangleArea(unsigned int length, unsigned int  
width);  
  
/*                                CalculateArea.cpp                                */  
unsigned int CalculateRectangleArea(unsigned int length, unsigned int  
width)  
{  
    return length * width;  
}
```

The steps for creating the "CalculateArea" library are as given in Figure 4.3 below.

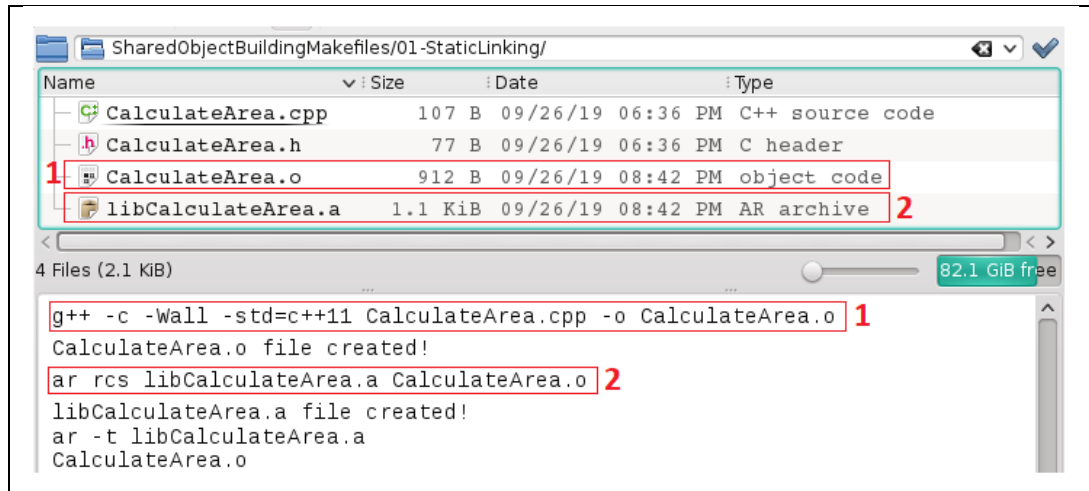


Figure 4.3: Static library generation.

Descriptions of compiler options given to g++ compiler are as follows [4]:

- c:

This flag prevents the object code file, which is created after compiling the source code, from being exported to the linker. Simply put, no linking is done. When this flag is used, the final output will be an object file for each source file.

After this flag is used, the names of the object files created by compiling the source files will be the same as the name of the source file. Only object code file extensions are converted to ".o". Unrecognized files are ignored during compilation.

- Wall:

This flag allows all warnings that occur during compilation to be displayed.

- std=c++11:

This flag allows the code to be compiled according to C++11 standards.

Descriptions of options given to archive file generator are as follows:

- ar rcs:

This command creates the archive file.

- ar -t:

This command displays the object files in the archive file.

5. DYNAMIC LINKING

In computer science, the dynamic linker is an application provided with the operating system due to its low-level requirements. Dynamic Linking is done by the dynamic linker. The task of the dynamic linker is to link the shared libraries that the executable program needs at runtime. When the linking process is performed, the contents of the library such as variables, functions, etc. are copied from the nonvolatile memory to RAM so that executable program can use this contents like own contents. At the same time, the dynamic linker populates the branch table and relocates the pointers.

The linking process is commonly known as a process that performed to an executable at compile time. Contrary to popular belief, external shared libraries are loaded into the running process and bonded these libraries using a dynamic linker, a special part of the operating system.

The files that the dynamic linkers interact with vary from the operating system to operating system. Dll (Dynamic-link Library) in Windows, .so (Shared Object) in Linux and .dylib (Dynamic Library) file extension in macOS are used by the dynamic linker.

5.1. Dynamic Linking Example

To explain the Dynamic Linking process, a simple implementation of "strcmp" and "strlen" functions, which is one of the most important parts of the standard C library, is used. These implementations are gathered under a library called "MyStringOp".

5.1.1. Creating Object File For Using In The Shared Object

The following Table 5.1 shows the implementation of the "strcmp" function in the standard C library.

Table 5.1: "my_strcmp" function header and source file.

```
/*          my_strcmp.h          */
int my_strcmp(const char *leftString, const char *rightString);

/*          my_strcmp.cpp          */
int my_strcmp(const char *leftString, const char *rightString)
{
    for (; *leftString == *rightString; ++leftString, ++rightString) {
        if (*leftString == '\0') {
            return 0;
        }
    }

    return *leftString - *rightString;
}
```

The following Table 5.2 shows the implementation of the "strlen" function in the standard C library.

Table 5.2: "my_strlen" function header and source file.

```
/*          my_strlen.h          */
int my_strlen(const char *string);

/*          my_strlen.cpp          */
int my_strlen(const char *string)
{
    const char *character;
    for (character = string; *character; ++character);

    return character - string;
}
```

Object files, to be used when creating a shared object, are compiled from the source file as shown in Figure 5.1 below. In this step, it is necessary to use "-fPIC" flag for object file creation.

- fPIC:

This flag allows the code to be compiled according to position-independent code requirements of the shared object files. Shared object files may need to be run to different addresses for different processes. Position-independent code does not contain any special addresses, in this way processes that use the shared object library can run these codes at different addresses [4].

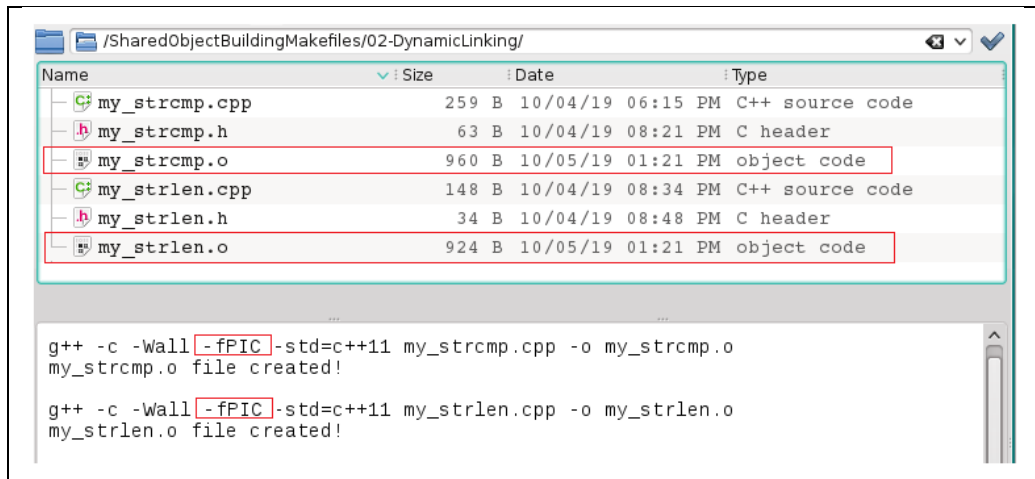


Figure 5.1: "my_strcmp" and "my_strlen" function output files generation.

5.1.2. Generation Shared Object Library

The "lib" prefix and the ".so" file extension are used to tell the compiler that the file is a library file, as mentioned in the previous sections.

The following Figure 5.2 shows how to generate a shared object library.

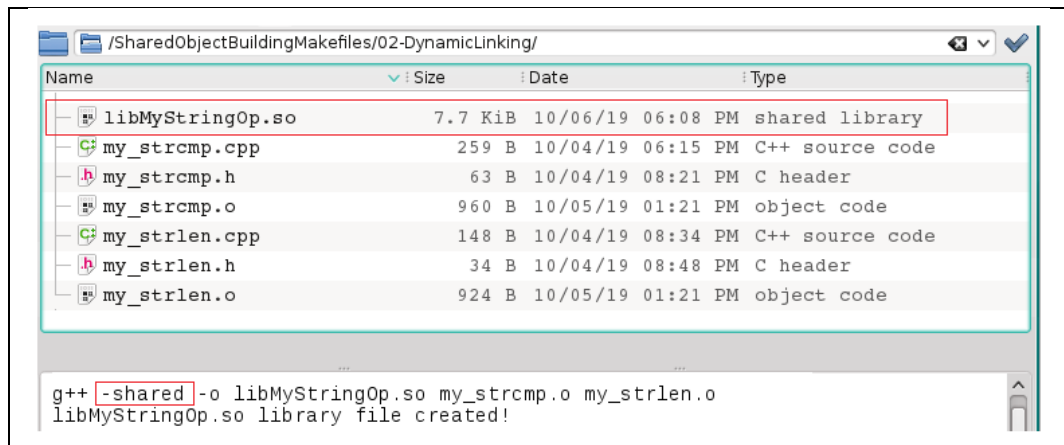


Figure 5.2: "MyStringOp" shared object library generation.

5.1.3. Compiling Test Program

The following Table 5.3 shows a simple test program for using the shared object library.

Table 5.3: Shared object library test program.

```
#include <iostream>                                     /* NOTE: std::cout */
#include "my_strcmp.h"
#include "my_strlen.h"

using namespace std;

int main(int argc, char **argv)
{
    string left = "Shared";
    string righth = "Object";

    if (!my_strcmp(left.c_str(), righth.c_str())) {
        cout << "Left and right strings are the same!" << endl;
    }
    else {
        cout << "Different string input!" << endl;
    }

    cout << "Character length of the left: " << my_strlen(left.c_str());
    cout << endl << endl;

    exit(EXIT_SUCCESS);
}
```

When the test program and the library file are in the same folder, attempting to compile fails as follows.

```
g++ -Wall -std=c++11 main.cpp -lMyStringOp -o prog.elf
...../...../i586-suse-linux/bin/ld: cannot find -lMyStringOp
collect2: error: ld returned 1 exit status
```

Figure 5.3: Compilation fails.

The reason for this error is that, by default, during the compilation phase, the compiler searches for "MyStringOp" library file in Linux's special library folders. To avoid this error, the path to the folder where the library is located during compilation must be pass to the compiler as a parameter. This is done in Figure 5.4 below and the compilation was successfully performed.

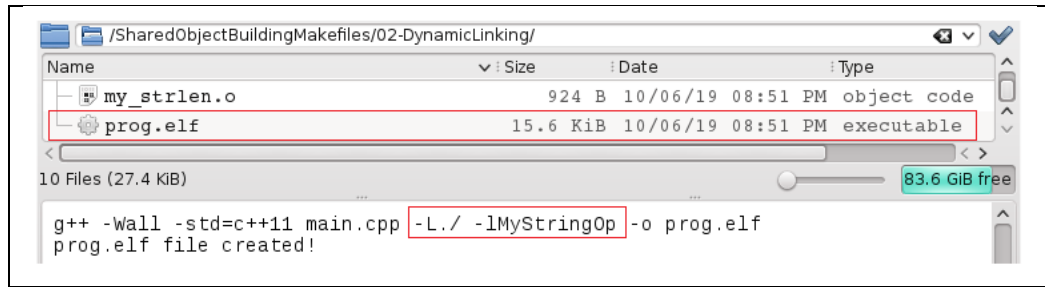


Figure 5.4: Compilation success.

The ldd console application is used to display which libraries required for "prog.elf" program [3].

```
02-DynamicLinking> ldd -r prog.elf
linux-gate.so.1 (0xb77d5000)
libMyStringOp.so => not found
libstdc++.so.6 => /usr/lib/libstdc++.so.6 (0xb76aa000)
libm.so.6 => /lib/libm.so.6 (0xb7664000)
libgcc_s.so.1 => /lib/libgcc_s.so.1 (0xb7647000)
libc.so.6 => /lib/libc.so.6 (0xb749a000)
/lib/ld-linux.so.2 (0xb77d8000)
undefined symbol: _Z9my_strlenPKc (./prog.elf)
undefined symbol: _Z9my_strcmpPKcS0_ (./prog.elf)
```

Figure 5.5: List dynamic dependencies.

As shown in Figure 5.5 above, the library path to which the "prog.elf" program depends is not known by the dynamic loader. This is observed as in Figure 5.6 below when trying to run the "prog.elf" program.

```
./prog.elf
./prog.elf: error while loading shared libraries:
libMyStringOp.so: cannot open shared object file:
No such file or directory
```

Figure 5.6: Dynamic loading error.

5.1.4. Unknown Library Path Problem Solutions

The reason for this error given in Figure 5.6 is that, by default, during the execution of the program, the dynamic loader searches for "MyStringOp" library file

in Linux's special library folders. To avoid this error, one of the following methods should be followed.

5.1.4.1. Rpath Option

The flag given in Figure 5.7 below is added during compilation to allow the dynamic loader to search the folder containing the program "prog.elf" during the execution. The program will work properly in this way.

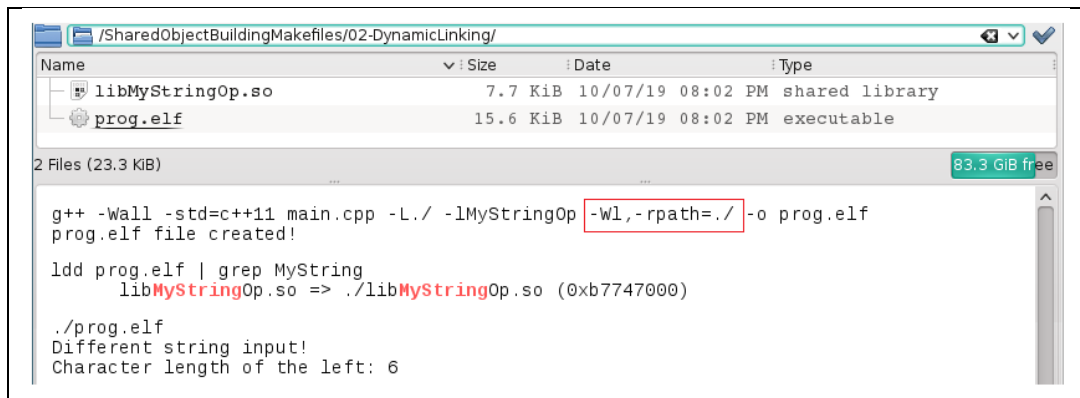


Figure 5.7: Compiling with "-rpath" option.

The disadvantage of this method is that the library file cannot be moved independently of the program and must be in the same folder as the program. In this case, the compiled version of the library for each program will be kept cloned in separate locations. All clone files will need to be changed when the library is modified.

If rpath is set to absolute path instead of relative path, all executable programs using this library search for the library file in the same location. However, in this case, executable programs will remain strictly dependent on the library path and the flexible working structure will be lost.

5.1.4.2. Changing LD_LIBRARY_PATH Value

`LD_LIBRARY_PATH` is an environment variable on Unix like operating systems. Environment variable contains the path list separated with colon characters. These paths used by the dynamic loader for searching shared libraries an application linked with.

This variable can be used for three main purposes;

- i) The testing a new version of a shared library with an already compiled executable program,
- ii) to use new shared libraries instead of the active ones in the system and to keep the old ones,
- iii) to prevent very large applications from using the libraries in the operating system and to provide all the libraries that the application is dependent on [5].

```
SharedObjectBuildingMakefiles/02-DynamicLinking> echo $LD_LIBRARY_PATH

./prog.elf
./prog.elf: error while loading shared libraries:
libMyStringOp.so: cannot open shared object file:
No such file or directory

SharedObjectBuildingMakefiles/02-DynamicLinking>
LD_LIBRARY_PATH= .../SharedObjectBuildingMakefiles/02-DynamicLinking:$LD_LIBRARY_PATH

SharedObjectBuildingMakefiles/02-DynamicLinking> echo $LD_LIBRARY_PATH
/media/$f_SharedFolder/Projects/MyProjects/SharedObjectBuildingMakefiles/02-DynamicLinking:

./prog.elf
./prog.elf: error while loading shared libraries:
libMyStringOp.so: cannot open shared object file:
No such file or directory
```

Figure 5.8: Non-exported environment variable.

It will be observed that although the `LD_LIBRARY_PATH` is changed as shown in Figure 5.8 above, the program cannot run properly. In Unix like operating systems, changes made to any environment variable will not be valid unless exported.

```
SharedObjectBuildingMakefiles/02-DynamicLinking> echo $LD_LIBRARY_PATH

./prog.elf
./prog.elf: error while loading shared libraries:
libMyStringOp.so: cannot open shared object file:
No such file or directory

SharedObjectBuildingMakefiles/02-DynamicLinking>
export LD_LIBRARY_PATH= .... /SharedObjectBuildingMakefiles/02-DynamicLinking:$LD_LIBRARY_PATH

SharedObjectBuildingMakefiles/02-DynamicLinking> echo $LD_LIBRARY_PATH
/media/sf_SharedFolder/Projects/MyProjects/SharedObjectBuildingMakefiles/02-DynamicLinking:

SharedObjectBuildingMakefiles/02-DynamicLinking> ldd prog.elf | grep MyString
libMyStringOp.so (libc6) => ..... /02-DynamicLinking/libMyStringOp.so (0xb7705000)

./prog.elf
Different string input!
Character length of the left: 6
```

Figure 5.9: Exported environment variable.

When the environment variable is exported as in Figure 5.9 above, the test program starts to run properly. It is inconvenient to use `LD_LIBRARY_PATH` except for the three reasons mentioned above.

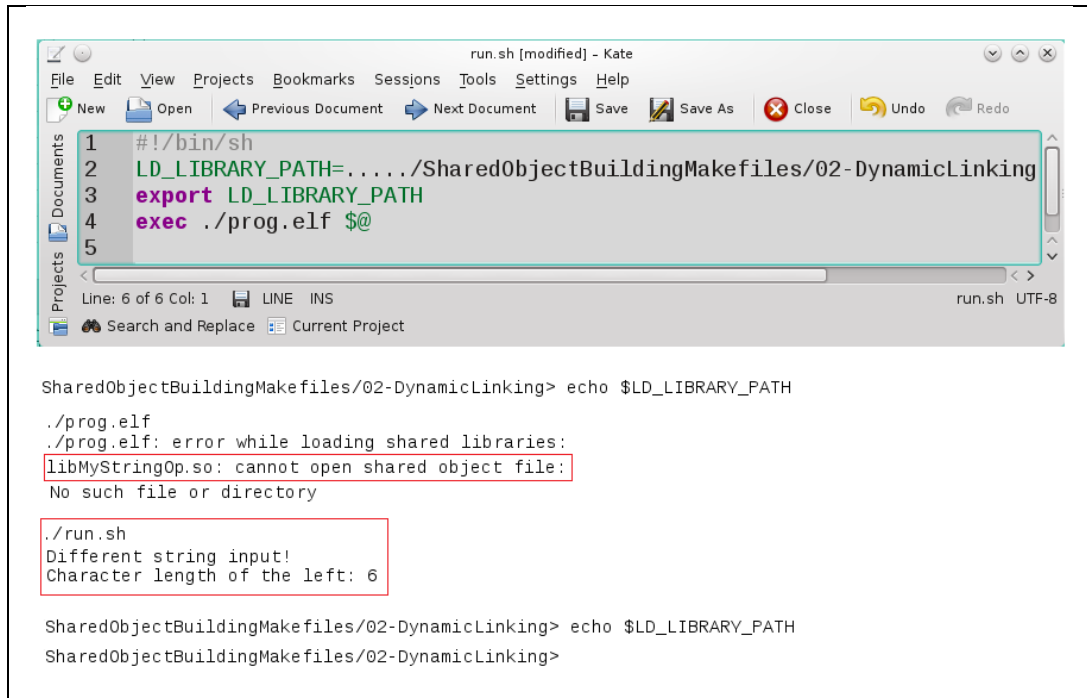
The following three problems can be encountered with the use of this variable;

- Malware can be injected into the application as a library,
- If a large directory is added to this variable, programs can slow down considerably due to system calls,
- A library that is not suitable for the application being run can be mistakenly used by the application, and in unpredictable situations, the application may crash.

`LD_LIBRARY_PATH` environment variable usage can be avoided by using one of the following methods.

- If you compile the application yourself, you can give the correct location of the library to the linker during compilation as shown in Figure 5.7.
- If you did not develop the application, there is a program called "chrpath" in Linux where you can change the rpath option in the application.
- If you cannot fix the executable program, you can open a subshell and modify the environment variable in this shell to make the changes only apply to the

program you want to run. This change will only take effect within the subshell and as long as the program is running.



```
run.sh [modified] - Kate
File Edit View Projects Bookmarks Sessions Tools Settings Help
New Open Previous Document Next Document Save Save As Close Undo Redo
1 #!/bin/sh
2 LD_LIBRARY_PATH=.../SharedObjectBuildingMakefiles/02-DynamicLinking
3 export LD_LIBRARY_PATH
4 exec ./prog.elf $@
5

Line: 6 of 6 Col: 1 LINE INS
run.sh UTF-8

SharedObjectBuildingMakefiles/02-DynamicLinking> echo $LD_LIBRARY_PATH
./prog.elf
./prog.elf: error while loading shared libraries:
libMyStringOp.so: cannot open shared object file:
No such file or directory

./run.sh
Different string input!
Character length of the left: 6

SharedObjectBuildingMakefiles/02-DynamicLinking> echo $LD_LIBRARY_PATH
SharedObjectBuildingMakefiles/02-DynamicLinking>
```

Figure 5.10: Running the executable on subshell (wrapper script).

Testing the program in the command line is a simple solution for testing. Global `LD_LIBRARY_PATH` environment variable does not affect this. This is done as in Figure 5.11 below.



```
SharedObjectBuildingMakefiles/02-DynamicLinking> echo $LD_LIBRARY_PATH
./prog.elf
./prog.elf: error while loading shared libraries:
libMyStringOp.so: cannot open shared object file:
No such file or directory

SharedObjectBuildingMakefiles/02-DynamicLinking>
env LD_LIBRARY_PATH=.../SharedObjectBuildingMakefiles/02-DynamicLinking:$LD_LIBRARY_PATH ./prog.elf
Different string input!
Character length of the left: 6

SharedObjectBuildingMakefiles/02-DynamicLinking> echo $LD_LIBRARY_PATH
SharedObjectBuildingMakefiles/02-DynamicLinking>
```

Figure 5.11: Testing in the command line.

Do not apply the method in Figure 5.9. Because it pollutes the global environment variable, other programs that use this variable will be affected and may

not be able to function properly. All the reasons why LD_LIBRARY_PATH is bad are described in reference [6].

5.1.4.3. Using ldconfig

In contrast to the above methods, this approach would be appropriate when a library is intended to be used by more than one program.

There are two ways to do this;

- i) Putting the compiled program into one of Linux's default library location ("/usr/local/lib", "/usr/lib", "/lib" and etc.). This method is dangerous because another library can be accidentally deleted.

```
./prog.elf
./prog.elf: error while loading shared libraries:
libMyStringOp.so: cannot open shared object file:
No such file or directory

..../02-DynamicLinking> sudo cp libMyStringOp.so /usr/lib
root's password:
..../02-DynamicLinking> sudo chmod 0755 /usr/lib/libMyStringOp.so
..../02-DynamicLinking> sudo /sbin/ldconfig
..../02-DynamicLinking> sudo /sbin/ldconfig -p | grep MyString
      libMyStringOp.so (libc6) => /usr/lib/libMyStringOp.so

..../02-DynamicLinking> ldd prog.elf | grep MyString
      libMyStringOp.so (libc6) => /usr/lib/libMyStringOp.so (0xb76db000)

..../02-DynamicLinking> ./prog.elf
Different string input!
Character length of the left: 6
```

Figure 5.12: Putting in the standard library location.

As shown in Figure 5.12 above, library file "libMyStringOp.so" which is required for the executable programs must be copied to the "/usr/lib" folder. This file must have read permissions to be used by executable programs. This is provided by the "chmod" program. The ldconfig program is run once to allow the library to be indexed and used by the dynamic loader [7].

- ii) Declaring non-standard library locations that ldconfig should look at during dynamic loading with a file with the extension .conf in "/etc/ld.so.conf.d/" folder. This is done as in Figure 5.13 below.

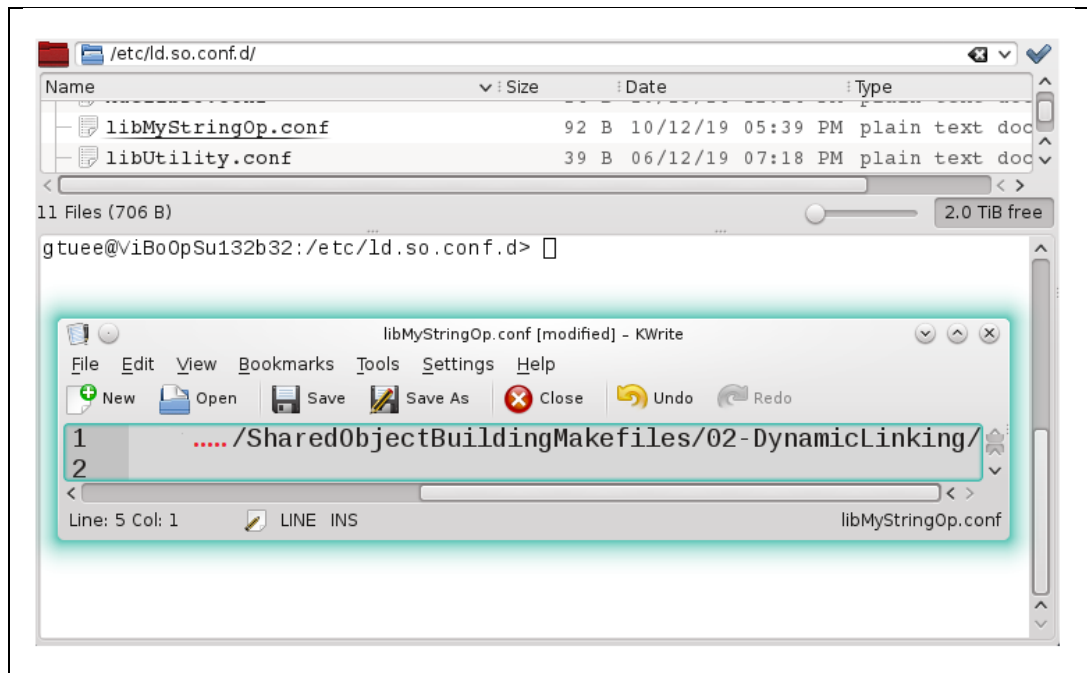


Figure 5.13: Adding "libMyStringOp.conf" file.

The ldconfig program is run once to allow the library to be indexed and used by the dynamic loader.

```

./prog.elf
./prog.elf: error while loading shared libraries:
libMyStringOp.so: cannot open shared object file:
No such file or directory

..../02-DynamicLinkinking> sudo /sbin/ldconfig
root's password:

..../02-DynamicLinkinking> sudo /sbin/ldconfig -p | grep MyString
libMyStringOp.so (libc6) => ..../02-DynamicLinkinking/libMyStringOp.so

..../02-DynamicLinkinking> ldd prog.elf | grep MyString
libMyStringOp.so (libc6) => ..../02-DynamicLinkinking/libMyStringOp.so (0xb7705000)

..../02-DynamicLinkinking> ./prog.elf
Different string input!
Character length of the left: 6

```

Figure 5.14: Putting in the non-standard library location.

6. DYNAMIC LOADING

Dynamic loading is a mechanism in which a computer program can load a library into memory at runtime, get addresses of functions and variables contained in the library, run these functions or access these variables and free this library from memory [3].

In contrast with static linking and dynamic linking, dynamic loading allows an executable program to start in the lack of these libraries, to find available libraries, and to gain additional functionality.

6.1. Dynamic Loading Use Cases

Dynamic loading is the most commonly used technique in the implementation of software plug-ins.

Dynamic loading is a technique that allows the user to select the desired library from multiple libraries doing the same job.

Dynamic loading is a technique that just makes the specified symbols in the library or executable program file available to the target executable program. In this way, memory usage of the target executable program reduced.

6.2. Declerations of Dynamic Loading Functions

Dynamic loading functions on Unix-like systems are declared in the header file "dlfcn.h" as given in Table 6.1 below [3].

Table 6.1: dlfcn.h file.

```
int    dlclose(void * handle);
char  *dlerror(void);
void  *dlopen(const char * file, int mode);
void  *dlsym(void *restrict handle, const char *restrict name);
```

6.2.1. dlopen Function

```
void * dlopen (const char * file, int mode);
```

The "dlopen()" function, make the symbols (functions and data variables in the symbol table) in the library or executable program file available to the target executable program.

The "file" argument is a pathname for the library or executable program file.

The "mode" argument is used for the select how to operate with another dependent library or executable program contained in the loaded symbols [3].

A successful "dlopen()" function call must return a symbol table handle. The caller can use this handle on following calls to "dlsym()" and "dlclose()" [3].

The following Table 6.2 shows how to use the "dlopen()" function.

Table 6.2: "dlopen()" and "dlerror()" function usage.

```
#include <dlfcn.h>                                /* NOTE: Dynamic Loading functions */
using namespace std;
int main(int argc, char **argv)
{
    // Codes...

    dlerror();                                     /* NOTE: Clear any existing error */

    handle = dlopen("libm.so", RTLD_LAZY);
    if (!handle) {
        cerr << dlerror() << endl;
        exit(EXIT_FAILURE);
    }

    // Codes...
}
```

6.2.2. dlsym Function

```
void * dlsym (void *restrict handle, const char *restrict name);
```

The "dlsym()" function gets the address of a symbol (function or data variable) defined in the symbol table identified by the "handle" argument.

The "handle" argument is a symbol table handle returned from a call to "dlopen()". In order to use the "handle" argument with "dlsym()" call, it must not have been previously released with the "dlclose()" call.

The "name" argument must contain the symbol's name (function or data variable name) as a character string.

The return value from "dlsym()" function call, must be cast to a function pointer, in case the "name" argument is a function. In this case, the return value will be used for the call of the function. The return value from "dlsym()" function call, must be cast to a pointer, in case the "name" argument is a data variable. In this case, the return value will be used for the access the contents of the data variable [3].

The following Table 6.3 shows how to use the "dlsym()" function.

Table 6.3: dlsym and dlerror function usage.

```
#include <dlfcn.h>                                /* NOTE: Dynamic Loading functions */
using namespace std;
int main(int argc, char **argv)
{
    // Codes...

    dlerror();                                       /* NOTE: Clear any existing error */

    FPSine = (double (*)(double)) dlsym(handle, "sin");
    if ((error = dlerror()) != NULL) {
        cerr << error << endl;
        exit(EXIT_FAILURE);
    }

    FPSquareRoot = (double (*)(double)) dlsym(handle, "sqrt");
    if ((error = dlerror()) != NULL) {
        cerr << error << endl;
        exit(EXIT_FAILURE);
    }

    // Codes...
}
```

6.2.3. dlclose Function

```
int    dlclose (void * handle);
```

The "dlclose()" function call informs the operating system that the symbol table handle specified by the "handle" argument will no longer be used by the calling executable program [3].

The following Table 6.4 shows how to use the "dlclose()" function.

Table 6.4: "dlclose()" and "dlerror()" function usage.

```
#include <dlfcn.h>                                /* NOTE: Dynamic Loading functions */
using namespace std;

int main(int argc, char **argv)
{
    // Codes...

    /* NOTE: FPSine and FPSquareRoot function */
    /* pointers are used in this function */
    SineWaveGenerator();

    int errorCode = 0;
    if ((errorCode = dlclose(handle)) != 0) {
        cerr << "Failed to close \"handle\"!" << endl;
        cerr << "Error Code: " << errorCode << endl;
        exit(EXIT_FAILURE);
    }

    // Codes...
}
```

6.2.4. dlerror Function

```
char * dlerror (void);
```

The "dlerror()" function call must return character string, which is null-terminated and with no trailing (newline), that describes the last fault happened while dynamic linking processing. If no dynamic loading fault happened since the last "dlerror()" function call, new "dlerror()" function call must return NULL value [3].

The use of the "dlerror()" function is as shown in Table 6.2, Table 6.3 and Table 6.4 above.

6.3. Dynamic Loading Example

This section will show the use of the "sin" and "sqrt" functions, which are declared in the "math.h" and defined in the <cmath> file, in a program using the dynamic loading method.

In Linux, the <cmath> file is embedded in the "libm.so" shared object library file. Therefore Table 6.2 shows that the target program loaded the "libm.so" file symbols to own symbol table.

Table 6.3 shows getting of the address of the functions "sin" and "sqrt" from the symbol table with "dlsym()" function call. The FPSquareRoot and FPSine function pointers are used in the program instead of the "sqrt" and "sin" functions, respectively.

Table 6.4 shows how to close the symbol table with "dlclose()" function call.

6.3.1. Used Equation in Example Program

In the example program, the data (time and voltage value) required to plot the time-dependent graph of a 0 - 120 millisecond part of the sine signal of the mains supply, which is subject to electrical engineering, is generated.

$$v_{(t)} = v_{max} \sin(\omega t) \quad (6.1)$$

$$v_{max} = v_{rms} \sqrt{2} \quad \text{and} \quad \omega = 2\pi f \quad (6.2)$$

Equation (6.1) can be written in the form equation (6.3).

$$v_{(t)} = v_{rms} \sqrt{2} \sin(2\pi f t) \quad (6.3)$$

$$v_{rms} = 220 \text{ VAC} \quad \text{and} \quad f = 50 \text{ Hz} \quad (6.4)$$

Turkey's mains voltage and frequency values are given in equation (6.4). Therefore equation (6.3) can be written in the form equation (6.5).

$$v_{(t)} = 220\sqrt{2} \sin(100\pi t) \quad (6.5)$$

The mathematical equation of the data created by the program is given in equation (6.5).

Similar to the Figure 6.1 given below, the voltage values corresponding to time at every 1 millisecond intervals will be calculated by the example program and written in a text file [8].

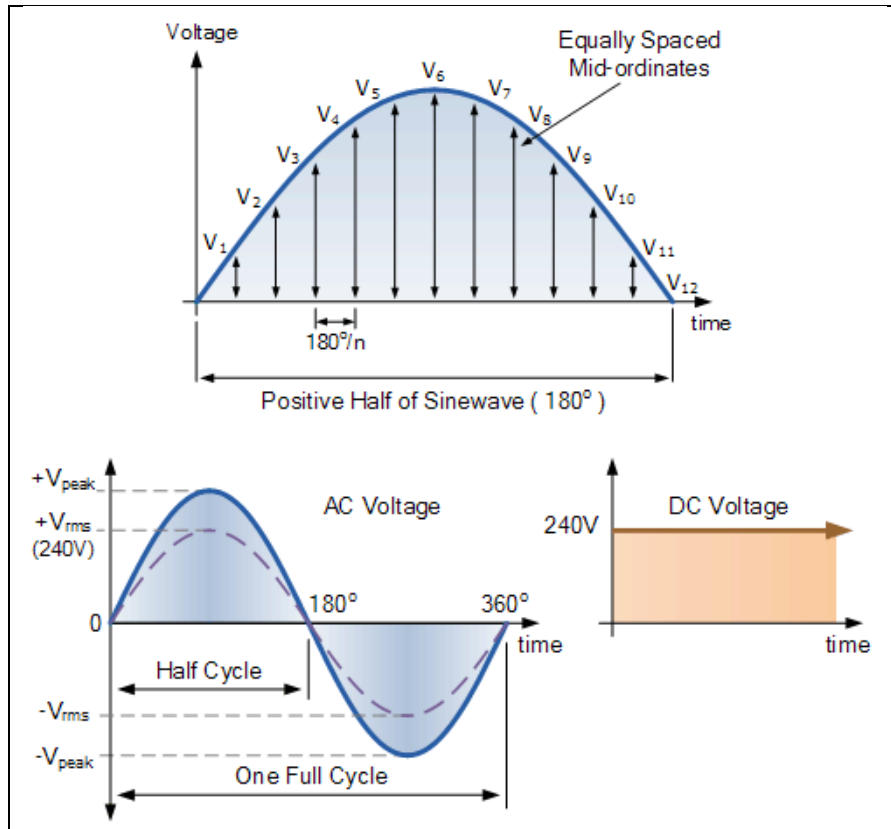


Figure 6.1: Voltage value at the specified time.

6.3.2. Generation and Plotting the Data

In the given Table 6.5 below, FPSquareRoot function pointer is used instead of "sqrt" function and FPSine function pointer is used instead of "sin" function.

Table 6.5: FPSquareRoot and FPSine function pointers usage.

```
void SineWaveGenerator()
{
    int frequency = 50;
    int RMSVoltageAmplitude = 220;
    double peakAmplitude = RMSVoltageAmplitude * FPSquareRoot(2);

    //
    // Create the data file.
    //
    string dataFileName = "sine_wave_timeline_data.txt";
```

Table 6.5: FPSquareRoot and FPSine function pointers usage.

```
ofstream data;

data.open(dataFileName.c_str());
for (int millisecond = 0; millisecond <= 120; ++millisecond)
{
    /* NOTE: The first row shows writing the time value to the file. */
    /* The third row shows writing the voltage value to the file. */
    data << fixed << setw(15) << setprecision(3) << 0.001 * millisecond
    << setw(22) << setprecision(7)
    << peakAmplitude * FPSine(2 * M_PI * frequency * 0.001 * millisecond)
    << endl;
}
data.close();

// Codes...
}
```

Some of the data generated by the program is as Figure 6.2 below.

sine_wave_timeline_data.txt ✕			sine_wave_timeline_data.txt ✕		
1	0.000	0.0000000	112	0.111	-96.1435254
2	0.001	96.1435254	113	0.112	-182.8758526
3	0.002	182.8758526	114	0.113	-251.7070172
4	0.003	251.7070172	115	0.114	-295.8993453
5	0.004	295.8993453	116	0.115	-311.1269837
6	0.005	311.1269837	117	0.116	-295.8993453
7	0.006	295.8993453	118	0.117	-251.7070172
8	0.007	251.7070172	119	0.118	-182.8758526
9	0.008	182.8758526	120	0.119	-96.1435254
10	0.009	96.1435254	121	0.120	-0.0000000

Figure 6.2: Generated data.

The generated data is drawn with "gnuplot" program as in Figure 6.3.

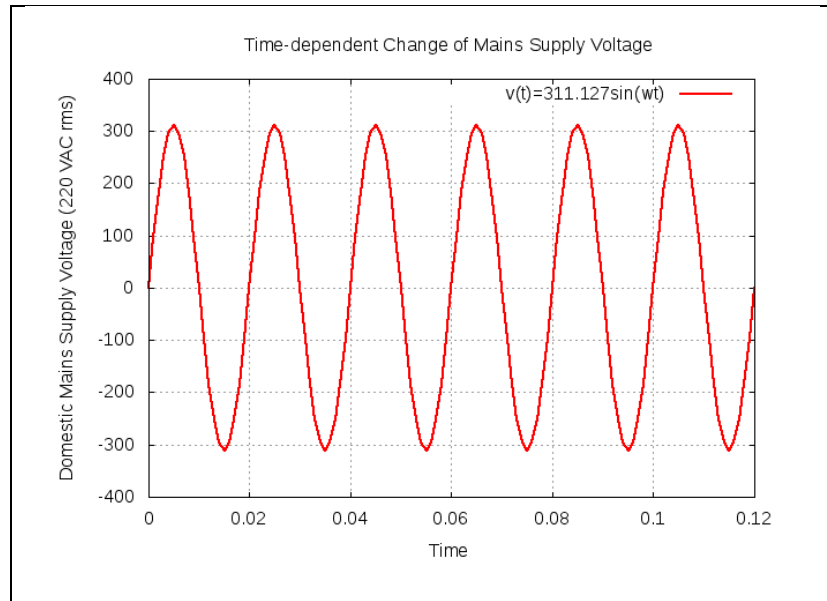


Figure 6.3: Time-dependent change of mains supply voltage.

6.3.3. Error Handling

In this section, the most basic errors that can occur in calls to dynamic loading functions and the printing of these errors with "dlerror()" function are explained.

If the "dlopen()" function given in Table 6.2 is called accidentally with the "libmn.so" parameter instead of "libm.so", which contains mathematical functions, the error given in Figure 6.4 occurs.

```
SharedObjectBuildingMakefiles/DynamicLoading> ./03-obj/prog.elf
libmn.so: cannot open shared object file: No such file or directory
```

Figure 6.4: Incorrect "dlopen()" call.

If the "dlsym()" function given in Table 6.3 is called accidentally with the "sinn" parameter instead of "sin", which calculates sine wave, the error given in Figure 6.5 occurs.

```
SharedObjectBuildingMakefiles/DynamicLoading> ./03-obj/prog.elf  
/lib/libm.so.6: undefined symbol: sinn
```

Figure 6.5: Incorrect "dlsym()" call.

If the "dlsym()" function given in Table 6.3 is called accidentally with the "sqrty" parameter instead of "sqrt", which calculates square root, the error given in Figure 6.6 occurs.

```
SharedObjectBuildingMakefiles/DynamicLoading> ./03-obj/prog.elf  
/lib/libm.so.6: undefined symbol: sqrty
```

Figure 6.6: Incorrect "dlsym()" call.

If the "dlclose()" function given in Table 6.4 is called accidentally with the uninitialized handle parameter instead of a valid handle, which initialized with "dlopen()" function call, the error given in Figure 6.7 occurs.

```
SharedObjectBuildingMakefiles/DynamicLoading> ./03-obj/prog.elf  
Inconsistency detected by ld.so: dl-close.c: 762: _dl_close:  
Assertion 'map->l_init_called' failed!
```

Figure 6.7: Incorrect "dlclose()" call.

If the "dlclose()" function given in Table 6.4 is called accidentally with the "NULL" parameter instead of a valid handle, which initialized with "dlopen()" function call, the error given in Figure 6.8 occurs.

```
SharedObjectBuildingMakefiles/DynamicLoading> ./03-obj/prog.elf  
Segmentation fault
```

Figure 6.8: Incorrect "dlclose()" call.

7. SHARED OBJECT BUILDING MAKEFILE

Within the scope of this thesis, 2 different makefiles were made. These makefiles are configured to compile projects and libraries.

These makefiles have been tried on Ubuntu and OpenSUSE operating systems and have been observed to be working well.

7.1. Library Makefile

With library makefile, source files to be used in multiple projects are compiled and collected under a single .so file. In this way, the re-insertion of these source files in each project was prevented and the editing processes that would occur due to the change of source files were reduced and the maintenance of the libraries was facilitated. The users of the libraries have been isolated from the exchange processes in the library and these changes have been made in the background by the authors of the library.

The following Figure 7.1 illustrates the operation of the generic makefile.

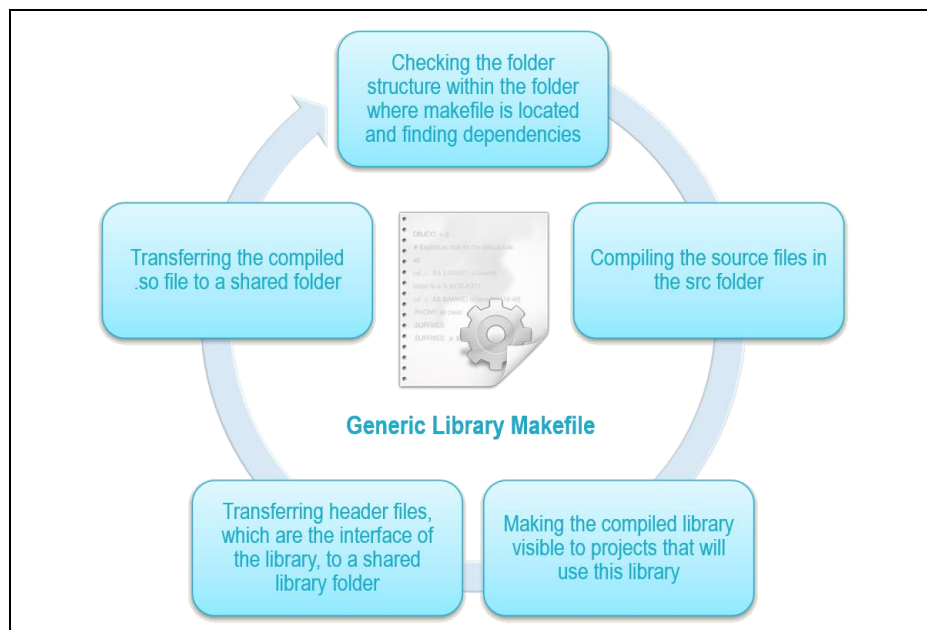


Figure 7.1: Generic library makefile.

The following Figure 7.2 shows the folder structure created for compiling library files. This folder structure should be created by library makefile users. The folder

structure is checked by the makefile. If there is no proper folder structure, the compilation process is terminated.

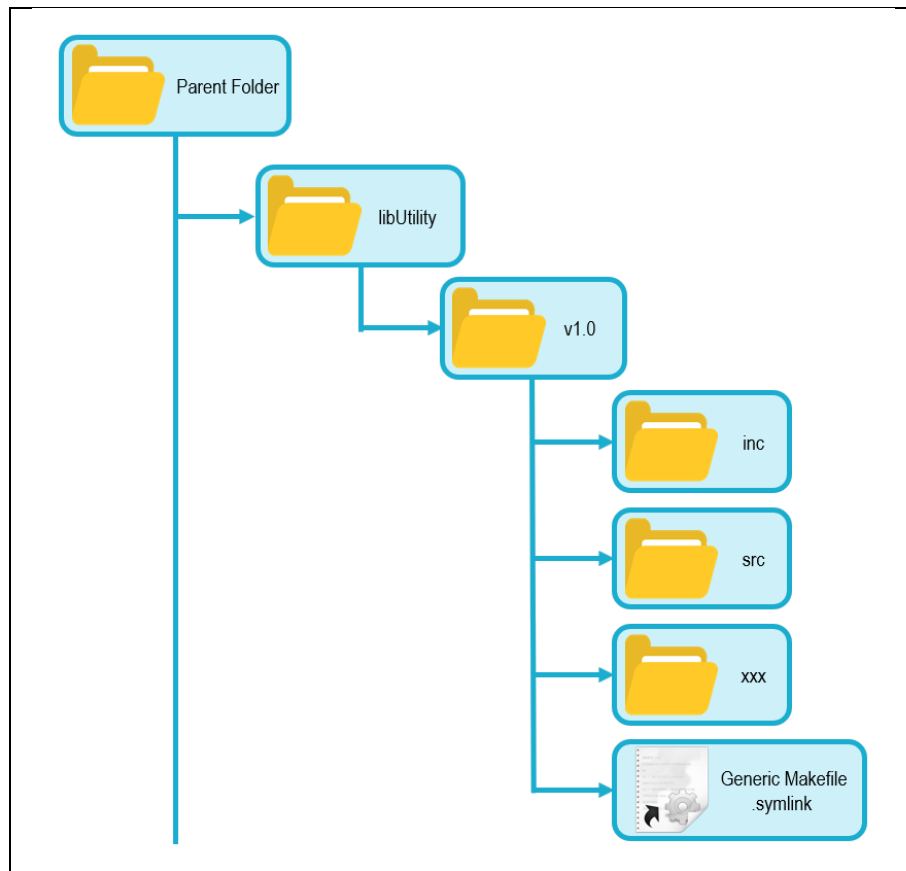


Figure 7.2: Library folder hierarchy.

7.1.1. Library Folder Detection

When the makefile is called, it looks at the 2 upper folders of the folder it is called from. If an incorrect library folder name is found during these checks, the compilation process is terminated with an error message.

This folder must have the "lib" prefix. If the naming of this folder does not contain the "lib" prefix, compilation terminated. The following Table 7.1 shows the code snippet from which this operation was performed.

Table 7.1: Library folder detection code.

```
LIB_NAME      = $(notdir $(shell dirname "${CURDIR}"))
PROJECT_NAME  = $(subst ${LIB_TAG},${empty},${LIB_NAME})

# Return space added value. For example "libPoissonPointProcess" to
# "lib PoissonPointProcess". Or return original value "Myproject" to "Myproject"
SPACE_ADDED_LIB_NAME = $(subst ${LIB_TAG},${LIB_TAG}${space},${LIB_NAME})

# New line for readable console output
$(info )

ifeq ($(word 1,${SPACE_ADDED_LIB_NAME}),${LIB_TAG})
    $(info Current parent folder "${LIB_NAME}" is a library folder.)
else
    $(error
    $(newline) Current parent folder "${LIB_NAME}" is not a library folder.
    $(newline) Library folder name must contain "lib" prefix!
    $(newline) Example: "libPoissonPointProcess"
    )
endif

# New line for readable console output
$(info )
```

Figure 7.3 below shows the error message due to an invalid library folder naming.

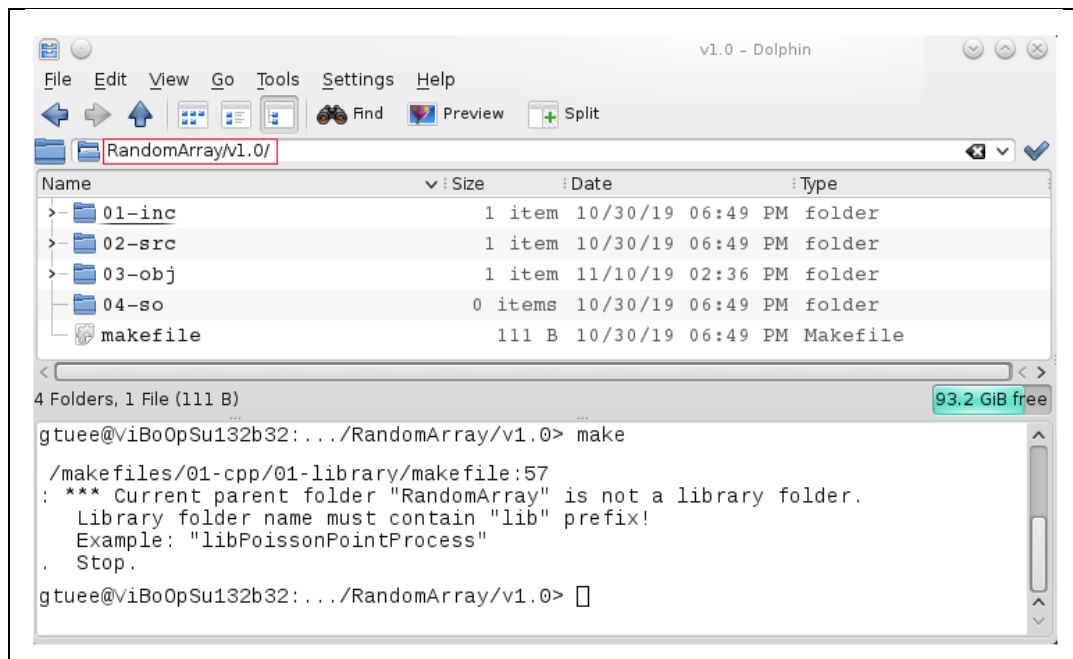


Figure 7.3: Invalid library folder detection.

Figure 7.4 below shows the information message given when compiling a library with a valid library folder naming.

```
gtuee@ViBoOpSu132b32:.../libRandomArray/v1.0> make
Current parent folder "libRandomArray" is a library folder.
gtuee@ViBoOpSu132b32:.../libRandomArray/v1.0> □
```

Figure 7.4: Valid library folder detection.

7.1.2. Library Version Folder Detection

When the makefile is called, it checks the naming of the folder it is in. If an incorrect version folder name is found during these checks, the compilation process is terminated with an error message.

This folder must have the "v" prefix. If the naming of the folder does not contain the "v" prefix, compilation terminated. The following Table 7.2 shows the code snippet from which this operation was performed.

Table 7.2: "v" prefix control code.

```
# result: vX.Y or invalid folder name
CURRENT_DIR = $(notdir $(CURDIR))

# Return space added value. For example "v1.0" to "v 1.0".
# Or return original value "y.e" to "y.e"
SPACE_ADDED_CURRENT_DIR = $(subst $(VER_TAG),$(VER_TAG)$(space),$(CURRENT_DIR))

ifneq ($(word 1,$(SPACE_ADDED_CURRENT_DIR)),$(VER_TAG))
    $(error \
$(newline) Current folder "$(CURRENT_DIR)" is not a library version folder.\
$(newline) Library version folder name must contain "v" prefix! \
$(newline) Example: "v1.3" or "v2.8" \
$(newline)
    )
endif
```

Figure 7.5 below shows what happens when the makefile is run in the folder where the version "v" prefix is not entered.

```
gtuee@ViBoOpSu132b32:.../libRandomArray/1.0> make
/makefiles/01-cpp/01-library/makefile:78
: *** Current folder "1.0" is not a library version folder.
    Library version folder name must contain "v" prefix!
    Example: "v1.3" or "v2.8"
    . Stop.
gtuee@ViBoOpSu132b32:.../libRandomArray/1.0> □
```

Figure 7.5: "v" prefix control output.

This folder must have a version number. If the naming of the folder does not contain the version number, compilation terminated. Table 7.3 below shows the code snippet in which this operation is performed.

Table 7.3: Version number empty control code.

```
# Result: X.Y or empty value
VERSION      = $(word 2,$(SPACE_ADDED_CURRENT_DIR))

ifeq ($(VERSION),$(empty))
    $(error \
$(newline) Current folder "$(CURRENT_DIR)" is not a library version folder.\
$(newline) Library version folder name must contain major and minor \
$(newline) version number separated by dot! \
$(newline) Example: "v1.3" or "v2.8" \
$(newline) \
    )
endif
```

Figure 7.6 below shows what happens when the makefile is run in the folder where the version number is not entered.

```
gtuee@ViBoOpSu132b32:.../libRandomArray/v> make
/makefiles/01-cpp/01-library/makefile:92: ***
Current folder "v" is not a library version folder.
Library version folder name must contain major and minor
version number separated by dot!
Example: "v1.3" or "v2.8"
. Stop.
gtuee@ViBoOpSu132b32:.../libRandomArray/v> □
```

Figure 7.6: Version number empty control output.

This folder must have a "." character for separating the major and minor version numbers. If the naming of the folder does not contain the "." character, compilation

terminated. Table 7.4 below shows the code snippet in which this operation is performed.

Table 7.4: "." character presence control code.

```
# If VERSION is contains a dot return dot
# else return null
IS_INCLUDE_DOT = $(findstring $(dot),$(VERSION))

ifeq ($(IS_INCLUDE_DOT),$(dot))
    $(info Current folder name "$(CURRENT_DIR)" is contains dot!)
else
    $(error \
$(newline) Current folder "$(CURRENT_DIR)" is not a library version folder.\
$(newline) Library version folder name must contain dot for separating \
$(newline) major and minor version number! \
$(newline) Example: "v1.3" or "v2.8" \
$(newline) \
)
endif
```

Figure 7.7 below shows what happens when the makefile is run in a folder without a "." character.

```
gtuee@ViBoOpSu132b32: .../libRandomArray/v12> make
/makefiles/01-cpp/01-library/makefile:108: ***
Current folder "v12" is not a library version folder.
Library version folder name must contain dot for separating
major and minor version number!
Example: "v1.3" or "v2.8"
. Stop.
gtuee@ViBoOpSu132b32: .../libRandomArray/v12> □
```

Figure 7.7: "." character presence control output.

This folder must have a valid major version number. If the naming of the folder does not contain the valid major version number, compilation terminated. Table 7.5 below shows the code snippet in which this operation is performed.

Table 7.5: Major version control code.

```
# Result: X Y or just X
SPACE_ADDED_VERSION_NAME = $(subst $(dot),$(space),$(VERSION))

# result: X
MAJOR_VER = $(word 1,$(SPACE_ADDED_VERSION_NAME))

IS_VALID_MAJOR_VER = $(shell [ $(MAJOR_VER) -ge 0 ] && echo true)

ifeq ($(IS_VALID_MAJOR_VER),true)
    $(info Current folder name "${CURRENT_DIR}" is contains valid major \
    version number "${MAJOR_VER}")
else
    $(error \
    $(newline) Major version number "${MAJOR_VER}" is not a \
    valid version number. \
    $(newline) Library version folder name must contain \
    valid major version number. \
    $(newline) Example: "v1.3" or "v2.8" \
    $(newline) \
    )
endif
```

Figure 7.8 below shows what happens when the makefile is run in the folder where the major version number is invalid.

```
gtuee@ViBoOpSu132b32:~/libRandomArray/v12x.1> make
/makefiles/01-cpp/01-library/makefile:129: ***
Major version number "12x" is not a valid version number.
Library version folder name must contain valid major version number.
Example: "v1.3" or "v2.8"
. Stop.
gtuee@ViBoOpSu132b32:~/libRandomArray/v12x.1> make
```

Figure 7.8: Major version control output.

This folder must have a valid minor version number. If the naming of the folder does not contain the valid minor version number, compilation terminated. Table 7.6 below shows the code snippet in which this operation is performed.

Table 7.6: Minor version control code.

```
# result: Y
MINOR_VER = ${word 2,${SPACE_ADDED_VERSION_NAME}}

IS_VALID_MINOR_VER = ${shell [ $(MINOR_VER) -ge 0 ] && echo true}

ifeq ($(IS_VALID_MINOR_VER),true)
    ${info} Current folder name "${CURRENT_DIR}" is contains valid minor \
    version number "${MINOR_VER}"!)
else
    ${error} \
    ${newline} Minor version number "${MINOR_VER}" is not a \
    valid version number. \
    ${newline} Library version folder name must contain \
    valid minor version number. \
    ${newline} Example: "v1.3" or "v2.8" \
    ${newline} \
    )
endif

# New line for readable console output
${info} )
```

Figure 7.9 below shows what happens when the makefile is run in the folder where the minor version number is invalid.

```
gtuee@ViBoOpSu132b32: .../libRandomArray/v12.1a> make
/makefiles/01-cpp/01-library/makefile:148: ***
Minor version number "1a" is not a valid version number.
Library version folder name must contain valid minor version number.
Example: "v1.3" or "v2.8"
. Stop.
gtuee@ViBoOpSu132b32: .../libRandomArray/v12.1a> □
```

Figure 7.9: Minor version control output.

If everything goes well in the library version folder check, the compilation process is continued. Figure 7.10 below shows the output if the makefile is run under a valid version folder.

```
gtuee@ViBoOpSu132b32: .../libRandomArray/v1.0> make
Current parent folder "libRandomArray" is a library folder.
Current folder name "v1.0" is contains dot!
Current folder name "v1.0" is contains valid major version number "1"!
Current folder name "v1.0" is contains valid minor version number "0"!
gtuee@ViBoOpSu132b32: .../libRandomArray/v1.0> █
```

Figure 7.10: Valid library folder detection.

7.1.3. Rules

2.1 Rules Of Makefile section describes what the rule is. There are 2 rules in this makefile. The first of these enables the shared object library project to be compiled. The second one is used to delete this compiled library.

The first rule is the rule that compiles the shared object library as given in Table 7.7 below. After executing this rule, the shared object library file and its symbolic links are created. These files are moved to the folder specified previously by the variable `CPP_COMPILED_LIBRARIES_SO_DIR`.

Table 7.7: Rule library.

```
library:
$(CPP) $(CPP_FLAGS) $(SRCS_CPP) $(INCLUDE_DIRS) $(USER_INC_DIR)
mv $(ROOT)/$(OBJ_NAME) $(OBJ_DIR)
$(CPP) -shared -Wl,-soname,$(LIB_NAME).so.$(MAJOR_VER) -o $(SO_NAME) $(OBJS_CPP)
cp $(INC_DIR)/$(HEADER_NAME) $(CPP_COMPILED_LIBRARIES_INC_DIR)
mv $(SO_NAME) $(CPP_COMPILED_LIBRARIES_SO_DIR)
ln -sf $(CPP_COMPILED_LIBRARIES_SO_DIR)/$(SO_NAME) $(SO_SYM_LNK_1)
ln -sf $(CPP_COMPILED_LIBRARIES_SO_DIR)/$(SO_NAME) $(SO_SYM_LNK_2)
sudo ldconfig
```

By executing the rule, a console output as in Figure 7.11 is obtained.

```
gtuee@viBoOpSu132b32:~/libRandomArray/v1.0> make library
CPP_LIBRARY_MAKEFILE_PATH path is:
"/media/sf_SharedFolder/Projects/MyProjects/SharedObjectBuildingMakefiles/makefiles/01-cpp/01-library/makefile"
Current parent folder "libRandomArray" is a library folder.
Current folder name "v1.0" is contains dot!
Current folder name "v1.0" is contains valid major version number "1"!
Current folder name "v1.0" is contains valid minor version number "0"!

g++ -Wall -fPIC -c -I./01-inc -std=c++11 -w ./02-src/RandomArray.cpp
-I/home/gtuee/SharedObjectBuildingMakefiles/CompiledLibraries/01-inc -Ixxx

mv ./RandomArray.o ./03-obj
g++ -shared -Wl,-soname,libRandomArray.so.1 -o libRandomArray.so.1.0 ./03-obj/RandomArray.o
cp ./01-inc/RandomArray.hpp /home/gtuee/SharedObjectBuildingMakefiles/CompiledLibraries/01-inc
mv libRandomArray.so.1.0 /home/gtuee/SharedObjectBuildingMakefiles/CompiledLibraries/02-so
ln -sf /home/gtuee/SharedObjectBuildingMakefiles/CompiledLibraries/02-so/libRandomArray.so.1.0
/home/gtuee/SharedObjectBuildingMakefiles/CompiledLibraries/02-so/libRandomArray.so.1
ln -sf /home/gtuee/SharedObjectBuildingMakefiles/CompiledLibraries/02-so/libRandomArray.so.1.0
/home/gtuee/SharedObjectBuildingMakefiles/CompiledLibraries/02-so/libRandomArray.so

sudo ldconfig
gtuee@viBoOpSu132b32:~/libRandomArray/v1.0> █
```

Figure 7.11: Rule library output.

The second rule is the rule that removes the shared object library from the installation folder as given in Table 7.8 below.

Table 7.8: Rule clean.

```
.PHONY: clean
clean:
-rm $(CPP_COMPILED_LIBRARIES_INC_DIR)/$(HEADER_NAME)
-rm -rf $(OBJ_DIR)/* $(CPP_COMPILED_LIBRARIES_SO_DIR)/$(SO_NAME) $(SO_SYM_LNK_1) $(SO_SYM_LNK_2)
sudo ldconfig
```

By executing the rule, a console output as in Figure 7.12 is obtained.

```

gtuee@ViBoOpSu132b32:~/libRandomArray/v1.0> make clean
CPP_LIBRARY_MAKEFILE_PATH path is:
"/media/sf_SharedFolder/Projects/MyProjects/SharedObjectBuildingMakefiles/makefiles/01-cpp/01-library/makefile"

Current parent folder "libRandomArray" is a library folder.

Current folder name "v1.0" is contains dot!
Current folder name "v1.0" is contains valid major version number "1"!
Current folder name "v1.0" is contains valid minor version number "0"!

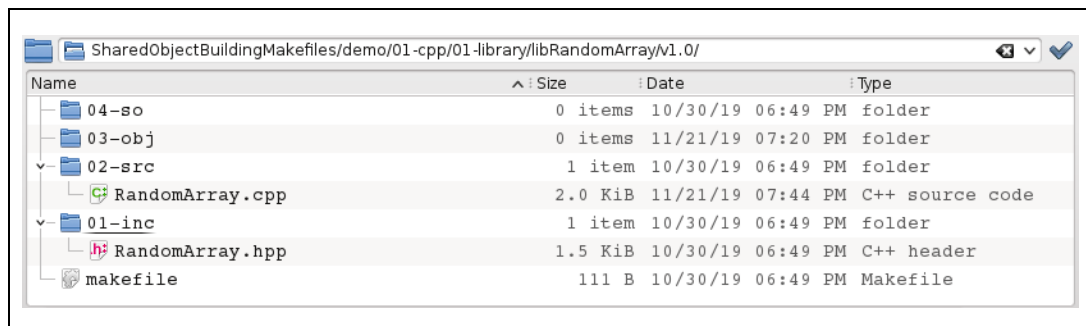
rm /home/gtuee/SharedObjectBuildingMakefiles/CompiledLibraries/01-inc/RandomArray.hpp
rm -rf ./03-obj/*
rm -rf /home/gtuee/SharedObjectBuildingMakefiles/CompiledLibraries/02-so/libRandomArray.so.1.0
rm -rf /home/gtuee/SharedObjectBuildingMakefiles/CompiledLibraries/02-so/libRandomArray.so.1
rm -rf /home/gtuee/SharedObjectBuildingMakefiles/CompiledLibraries/02-so/libRandomArray.so
sudo ldconfig
gtuee@ViBoOpSu132b32:~/libRandomArray/v1.0> █

```

Figure 7.12: Rule clean output.

7.2. Library Makefile Demo Project

In this section, a demo project is prepared which can be compiled with makefile which is explained in detail in section 7.1. The folder structure of this library project and the files stored in it are as given in Figure 7.13.



Name	Size	Date	Type
04-so	0 items	10/30/19 06:49 PM	folder
03-obj	0 items	11/21/19 07:20 PM	folder
02-src	1 item	10/30/19 06:49 PM	folder
RandomArray.cpp	2.0 KiB	11/21/19 07:44 PM	C++ source code
01-inc	1 item	10/30/19 06:49 PM	folder
RandomArray.hpp	1.5 KiB	10/30/19 06:49 PM	C++ header
makefile	111 B	10/30/19 06:49 PM	Makefile

Figure 7.13: Demo project folder structure.

As can be seen from the header file given in Table 7.9 below, the library fills random numbers into the array given as an argument to the FillRandomValue methods.

Table 7.9: RandomArray.hpp file.

```

* <p>
* This class implements the generating random array methods.
* </p>
*
* @author Önder Görmez
* @version 1.0
* @since December 01, 2018
*
*/
class Array

```

Table 7.9: RandomArray.hpp file.

```
{
private:
    static uint32_t InstanceCount;

public:
    Array();
    ~Array();
    static vector<uint32_t> &FillRandomValue(vector<uint32_t> &array);
    static void DisplayLibraryVersion();
};
```

The implementation of the library is as given in Table 7.10.

Table 7.10: RandomArray.cpp file.

```
#include <iostream>
#include <cstdlib>    /* srand, rand */
#include <ctime>      /* time */
#include "RandomArray.hpp"

using namespace std;

/*
 * Constructor
 */
Array::Array()
{
    // Codes...
}

/*
 * Destructer
 */
Array::~~Array()
{
    // Codes...
}

uint32_t Array::InstanceCount = 0;

/*
 * Fill Random Value
 */
vector<uint32_t> &Array::FillRandomValue(vector<uint32_t> &array)
{
#ifdef NDEBUG
    cout << endl;
    cout << "*****" << endl;
    cout << "Array::FillRandomValue function called!" << endl;
#endif
    /* initialize random seed */
    srand(time(NULL));
    for (uint32_t i = 0; i < array.size(); ++i) {
        array[i] = rand() % 200 + 1;
    }
}
```

Table 7.10: RandomArray.cpp file.

```

    }

#ifdef NDEBUG
    cout << "Array::FillRandomValue function call ended!" << endl;
    cout << "*****" << endl;
#endif
    return array;
};

```

When the project is compiled with generic makefiles written for libraries, the console output is as in Figure 7.14.

```

gtuee@ViBoOpSu132b32:~/libRandomArray/v1.0> make
CPP_LIBRARY_MAKEFILE_PATH path is: "/media/sf_SharedFolder/Projects/MyProjects/SharedObjectBuildingMakefiles/makefiles/01-cpp/01-library/makefile"

Current parent folder "libRandomArray" is a library folder.

Current folder name "v1.0" is contains dot!
Current folder name "v1.0" is contains valid major version number "1"!
Current folder name "v1.0" is contains valid minor version number "0"!

g++ -Wall -fPIC -c -I./01-inc -std=c++11 -w ./02-src/RandomArray.cpp -I/home/gtuee/SharedObjectBuildingMakefiles/CompiledLibraries/01-inc -Ixxx
mv ./RandomArray.o ./03-obj
g++ -shared -Wl,-soname,libRandomArray.so.1 -o libRandomArray.so.1.0 ./03-obj/RandomArray.o
cp ./01-inc/RandomArray.hpp /home/gtuee/SharedObjectBuildingMakefiles/CompiledLibraries/01-inc
mv libRandomArray.so.1.0 /home/gtuee/SharedObjectBuildingMakefiles/CompiledLibraries/02-so
ln -sf /home/gtuee/SharedObjectBuildingMakefiles/CompiledLibraries/02-so/libRandomArray.so.1.0 /home/gtuee/SharedObjectBuildingMakefiles/CompiledLibraries/02-so/libRandomArray.so.1
ln -sf /home/gtuee/SharedObjectBuildingMakefiles/CompiledLibraries/02-so/libRandomArray.so.1.0 /home/gtuee/SharedObjectBuildingMakefiles/CompiledLibraries/02-so/libRandomArray.so
sudo ldconfig
gtuee@ViBoOpSu132b32:~/libRandomArray/v1.0> █

```

Figure 7.14: Demo project compilation.

After compilation, the files required to use the library are copied to the environment variable paths specified during installation. Figure 7.15 shows the copied files of the random array library.

Name	Type	Size	Date
RandomArray.hpp	C++ header	1.5 KiB	11/21/19 07:20 PM
libRandomArray.so	shared library	12.8 KiB	11/21/19 07:20 PM
libRandomArray.so.1	shared library	12.8 KiB	11/21/19 07:20 PM
libRandomArray.so.1.0	shared library	12.8 KiB	11/21/19 07:20 PM

Figure 7.15: Demo project output files.

7.3. Project Makefile

With the project makefile, shared object library files created with the library makefile can be used easily. The output generated by the project makefile is an executable file. Thus, any researcher who uses this makefile performs his or her own work without having to deal with the details of the makefile design.

The following Figure 7.16 illustrates the operation of the generic makefile.

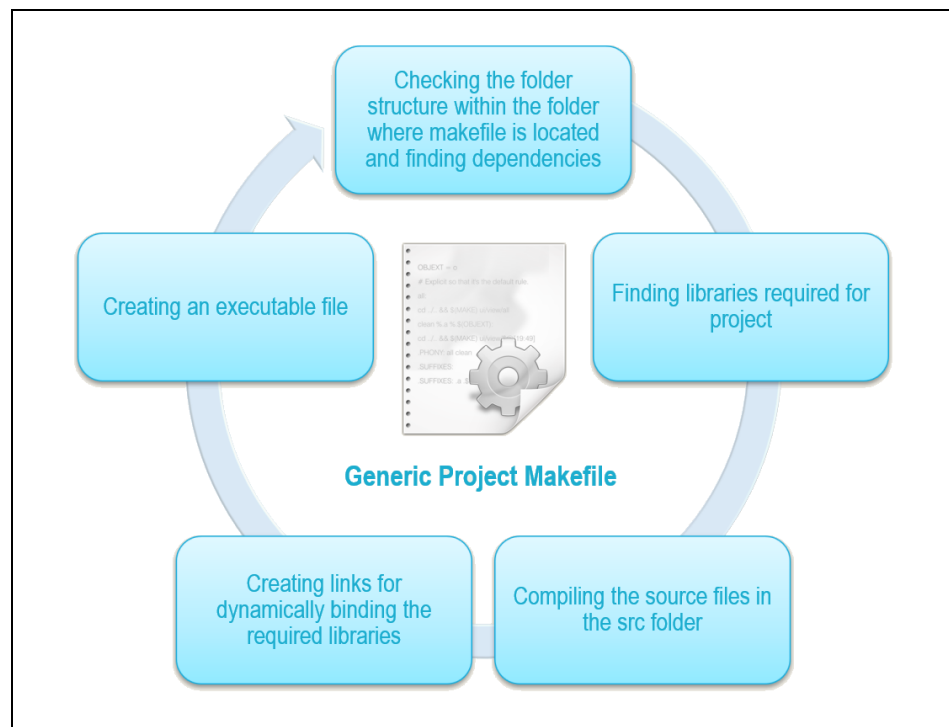


Figure 7.16: Generic project makefile.

The following Figure 7.17 shows the folder structure created for compiling project files. This folder structure should be created by project makefile users. The folder structure is checked by the makefile. If there is no proper folder structure, the compilation process is terminated.

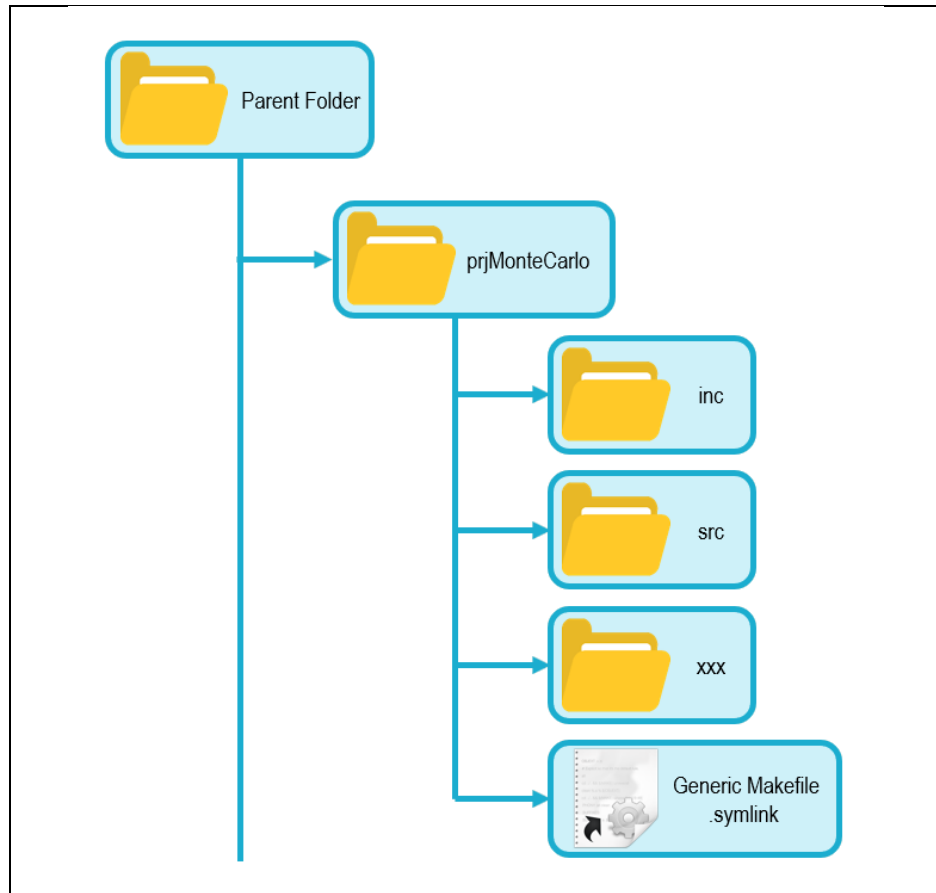


Figure 7.17: Project folder hierarchy.

7.3.1. Project Folder Detection

When the makefile is called, it looks at the parent folder of the folder it is called from. If an incorrect project folder name is found during these checks, the compilation process is terminated with an error message.

This folder must have the "prj" prefix. If the naming of this folder does not contain the "prj" prefix, compilation terminated. The following Table 7.11 shows the code snippet from which this operation was performed.

Table 7.11: "prj" prefix detection code.

```
FOLDER_NAME = $(notdir $(shell pwd))

# Return space added value. For example "prjSortAndDisplay" to
# "prj SortAndDisplay". Or return original value "Myproject" to "Myproject"
SPACE_ADDED_FOLDER_NAME = $(subst ${PROJECT_TAG},${PROJECT_TAG}${space}, \
                                ${FOLDER_NAME})

# New line for readable console output
$(info )

ifneq ($(word 1,${SPACE_ADDED_FOLDER_NAME}),${PROJECT_TAG})
    $(error \
    $(newline) Current folder "${FOLDER_NAME}" is not a project folder. \
    $(newline) Project folder name must contain "${PROJECT_TAG}" prefix! \
    $(newline) Example: "${PROJECT_TAG}SortAndDisplay" \
    $(newline) )
endif
```

Figure 7.18 below shows the error message due to an invalid project folder naming.

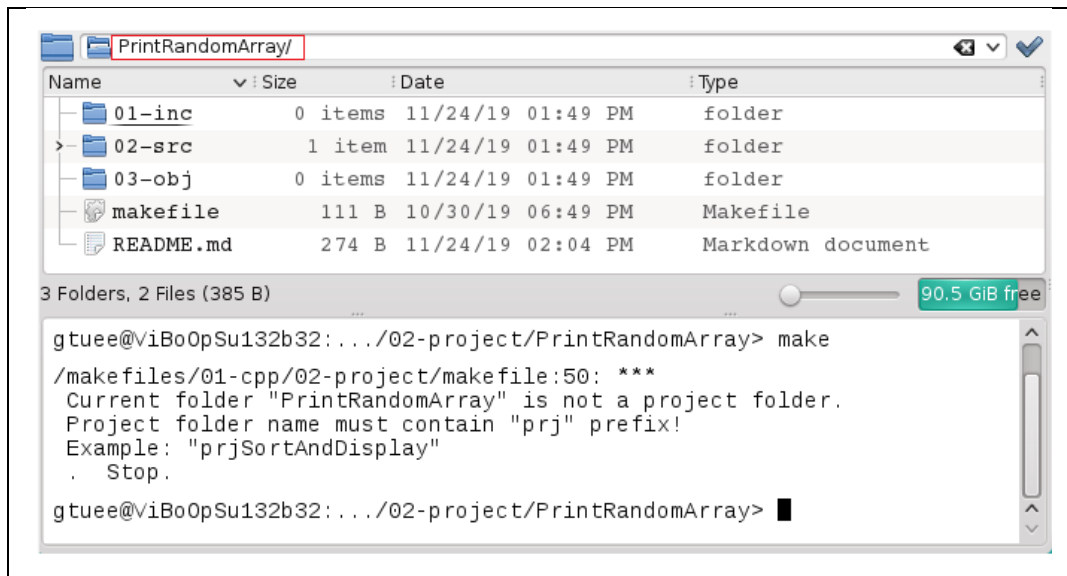


Figure 7.18: Invalid project folder detection.

This folder must have the project name after "prj" prefix. If the naming of this folder does not contain the project name after "prj" prefix, compilation terminated. The following Table 7.12 shows the code snippet from which this operation was performed.

Table 7.12: Project name detection code.

```
ifeq ($(PROJECT_NAME),$(empty))
    $(error
    $(newline) Current folder "${FOLDER_NAME}" is not a project folder.
    $(newline) Project folder name must contain project name after
    |         | "${PROJECT_TAG}" prefix!
    $(newline) Example: "${PROJECT_TAG}SortAndDisplay"
    $(newline)
    )
else
    $(info Current folder "${FOLDER_NAME}" is a project folder.)
endif

# New line for readable console output
$(info )
```

Figure 7.19 below shows the error message due to an invalid project folder naming.

```
gtuee@ViBoOpSu132b32: .../02-project/prj> make
/media/sf_SharedFolder/Projects/MyProjects/SharedObjectBuildingMakefiles
/makefiles/01-cpp/02-project/makefile:62: ***
Current folder "prj" is not a project folder.
Project folder name must contain project name after "prj" prefix!
Example: "prjSortAndDisplay"
. Stop.
gtuee@ViBoOpSu132b32: .../02-project/prj> □
```

Figure 7.19: Invalid project folder detection.

Figure 7.20 below shows the information message given when compiling a project with a valid project folder naming.

```
gtuee@ViBoOpSu132b32: ../02-project/prjPrintRandomArray> make
Current folder "prjPrintRandomArray" is a project folder.
gtuee@ViBoOpSu132b32: ../02-project/prjPrintRandomArray> □
```

Figure 7.20: Valid project folder detection.

7.3.2. Rules

2.1 Rules Of Makefile section describes what the rule is. There are 2 rules in this makefile. The first of these enables the project to be compiled. The second one is used to delete this compiled project.

The first rule is the rule that compiles the project as given in Table 7.13 below. After executing this rule, the project executable file is created.

Table 7.13: Rule project.

```
project:
    $(CPP) $(CPP_FLAGS) $(SRCS_CPP) \
    -I$(INC_DIR) -I$(CPP_COMPILED_LIBRARIES_INC_DIR) \
    -ldl -L$(CPP_COMPILED_LIBRARIES_SO_DIR) $(USED_LIBRARIES) \
    -o $(PROJECT_NAME).elf
```

By executing the rule, a console output as in Figure 7.21 is obtained.

```
gtuee@ViBoOpSu132b32: ../02-project/prjPrintRandomArray> make project
CPP_PROJECT_MAKEFILE_PATH path is:
"/media/sf_SharedFolder/Projects/MyProjects/SharedObjectBuildingMakefiles/makefiles/01-cpp/02-project/makefile"

Current folder "prjPrintRandomArray" is a project folder.

Shared object libraries in this system: RandomArray Sorting Utility

g++ -Wall -fPIC -std=c++11 -w ./02-src/main.cpp \
-I./01-inc -I/home/gtuee/SharedObjectBuildingMakefiles/CompiledLibraries/01-inc \
-ldl -L/home/gtuee/SharedObjectBuildingMakefiles/CompiledLibraries/02-so -lRandomArray -lSorting -lUtility \
-o PrintRandomArray.elf
gtuee@ViBoOpSu132b32: ../02-project/prjPrintRandomArray>
```

Figure 7.21: Rule project output.

The second rule is the rule that removes the project executable file as given in Table 7.14 below.

Table 7.14: Rule clean.

```
.PHONY: clean
clean:
    -rm -rf $(OBJ_DIR)/*
    -rm -rf $(PROJECT_NAME).elf
```

By executing the rule, a console output as in Figure 7.22 is obtained.

```
gtuee@ViBoOpSu132b32:~/02-project/prjPrintRandomArray> make clean
CPP_PROJECT_MAKEFILE_PATH path is:
"/media/sf_SharedFolder/Projects/MyProjects/SharedObjectBuildingMakefiles/makefiles/01-cpp/02-project/makefile"

Current folder "prjPrintRandomArray" is a project folder.

Shared object libraries in this system: RandomArray Sorting Utility

rm -rf ./03-obj/*
rm -rf PrintRandomArray.elf
gtuee@ViBoOpSu132b32:~/02-project/prjPrintRandomArray> █
```

Figure 7.22: Rule clean output.

7.4. Project Makefile Demo Project

In this section, a demo project is prepared which can be compiled with makefile which is explained in detail in section 7.3. The folder structure of this project and the files stored in it are as given in Figure 7.23.

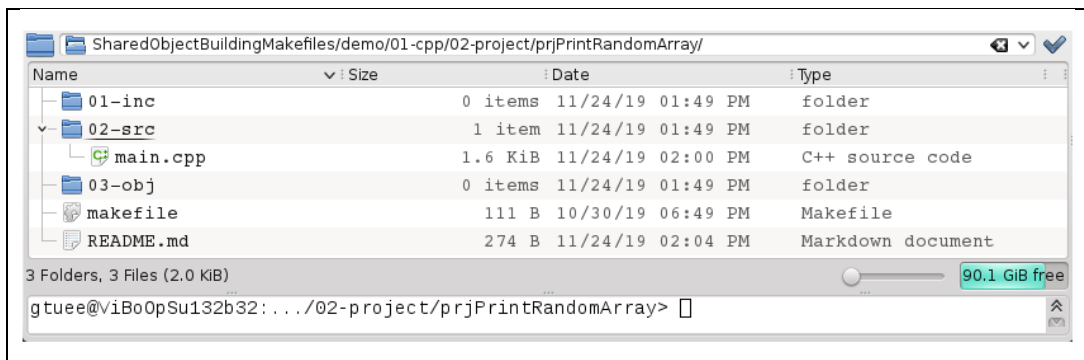


Figure 7.23: Demo Project Folder Structure

As can be seen from the main.cpp source file given in Table 7.15 below, using the library demo project output (libRandomArray.so) given as an example in section 7.2, a vector filled with random numbers will be created. This vector will be displayed in the console using the "DisplayVector" method.

Table 7.15: main.cpp file.

```
#include <iostream>
#include "RandomArray.hpp"

void DisplayVector(vector<uint32_t> &vectorToBeDisplayed);

using namespace std;

int main(void)
{
    cout << "This is a test project" << endl;

    Array::DisplayLibraryVersion();

    vector<uint32_t> testArray = vector<uint32_t>(7);
    Array::FillRandomValue(testArray);
    DisplayVector(testArray);

    return 0;
}

void DisplayVector(vector<uint32_t> &vectorToBeDisplayed)
{
    cout << endl;
    cout << "Size: " << vectorToBeDisplayed.size() << endl;
    for (uint32_t i = 0; i < vectorToBeDisplayed.size(); ++i) {
        cout << vectorToBeDisplayed[i] << " ";
    }
    cout << endl;
    cout << endl;
}
```

When the project is compiled with generic makefiles written for projects, the console output is as in Figure 7.24.

```
gtuee@ViBoOpSu132b32:~/02-project/prjPrintRandomArray> make
CXX_PROJECT_MAKEFILE_PATH path is: "/media/sf_SharedFolder/Projects/MyProjects
/SharedObjectBuildingMakefiles/makefiles/01-cpp/02-project/makefile"

Current folder "prjPrintRandomArray" is a project folder.

Shared object libraries in this system: RandomArray Sorting Utility

g++ -Wall -fPIC -std=c++11 -w ./02-src/main.cpp \
-I./01-inc -I/home/gtuee/SharedObjectBuildingMakefiles/CompiledLibraries/01-inc \
-ldl -L/home/gtuee/SharedObjectBuildingMakefiles/CompiledLibraries/02-so -lRandomArray -lSorting -lUtility \
-o PrintRandomArray.elf
gtuee@ViBoOpSu132b32:~/02-project/prjPrintRandomArray> █
```

Figure 7.24: Demo project compilation.

When the "PrintRandomArray.elf" file is executed, an output as in Figure 7.25 is obtained.

The screenshot shows a Dolphin IDE window titled "prjPrintRandomArray - Dolphin". The menu bar includes File, Edit, View, Go, Tools, Settings, and Help. Below the menu bar is a toolbar with icons for navigation and editing. The address bar shows the path "SharedObjectBuildingMakefiles/demo/01-cpp/02-project/prjPrintRandomArray/". A file explorer pane on the left lists two files: "makefile" (111 B, 10/30/19 06:49 PM, Makefile) and "PrintRandomArray.elf" (21.8 KiB, 12/07/19 03:02 PM, executable). The main console area shows the output of running the program. The prompt is "gtuee@ViBoOpSu132b32: ../02-project/prjPrintRandomArray> ./PrintRandomArray.elf". The output text is as follows:

```
This is a test project

*****
Random Array Library v1.0 used
libRandomArray.so.1.0 linked dynamically
*****

*****
Array::FillRandomValue function called!
Array::FillRandomValue function call ended!
*****

Size: 7
100 49 93 96 197 102 70

gtuee@ViBoOpSu132b32: ../02-project/prjPrintRandomArray> █
```

Figure 7.25: Demo project output.

8. CONCLUSION AND FUTURE WORK

8.1. Conclusion

In this thesis, makefile which is optimized for easy compilation and use of projects and libraries created in C/C++ languages on Unix-like operating systems was developed.

This makefile;

- Can recognize a library folder that located in any location on the computer, compile and link this library with other shared libraries that the library need,
- Shared objects version number can be created automatically according to the version number of the library folder,
- Can solve dependencies on source code and header files grouped as folders during the compilation and can compile and link the files according to these dependencies,
- It can automatically check whether the codes that are included in the project have been changed or not, during recompilation, only the source codes that depend on the changed files can be included in the compilation process,
- The library files (.so files) created after the compilation and linking operations and the header files, which is the interface of these library files, can be automatically collected under a folder that can be used by other projects.

Manually performing these automatic operations performed by makefile, causes serious time loss. Thanks to the generic nature of the Makefile project, all the code can be compiled with a single command and application files can be generated without the need to deal with low-level details of makefile during the compilation and linking of libraries. In this way, the duration of the studies is significantly reduced and researchers can focus on the details of their own research.

8.2. Future Work

The studies related to this thesis can be accessed through the github repo given in the references. Through this repo, interested people can contribute to the studies. In this way, a community is created around the subject and general solutions are provided for everyone [9].

Some suggestions for developing and making this thesis more general can be listed as follows;

- A top-level makefile can be written to compile multiple projects with a single make command.
- Existing makefiles can be expanded to compile source code in multiple folders during compilation.
- Demo projects about OpenCV, IT++, and Mex function for Octave can be created and added to the repo.
- Generic makefiles that serve different libraries and jobs can be added.
- Customized makefiles can be added to compile static libraries.

REFERENCES

- [1] Web 1, (2019), [https://en.wikipedia.org/wiki/Make_\(software\)](https://en.wikipedia.org/wiki/Make_(software)), (Date of Access: 20/10/2019)
- [2] Mecklenburg R., (2004), “Managing Projects with GNU Make”, 3rd Edition, O’Reilly Media, Inc.
- [3] Matsubara K., Kwok E., Rodriguez I., Paramasivam M., (2003), “Developing and Porting C and C++ Applications on AIX”, 2nd Edition, IBM Redbooks - IBM Corp.
- [4] Stallman R. M., GCC Developer Community, (2019), “Using the GNU Compiler Collection - for GCC Version 9.2.0”, GNU Press - Free Software Foundation, Inc.
- [5] Web 2, (2019), https://www.hpc.dtu.dk/?page_id=1180, (Date of Access: 07/10/2019)
- [6] Web 3, (2019), http://xahlee.info/UnixResource_dir/_ldpath.html, (Date of Access: 12/10/2019)
- [7] Web 4, (2019), <https://www.cprogramming.com/tutorial/shared-libraries-linux-gcc.html>, (Date of Access: 04/10/2019)
- [8] Web 5, (2019), <https://www.electronics-tutorials.ws/accircuits/rms-voltage.html>, (Date of Access: 15/09/2019)
- [9] Web 6, (2019), <https://github.com/ondergormez/SharedObjectBuildingMakefiles>, (Date of Access: 22/12/2019)
- [10] Stallman R. M., McGrath R., Smith P., (2002), “GNU Make - A Program for Directing Recompilation - for GNU make Version 3.80”, Free Software Foundation, Inc.
- [11] Kernighan B. W., Ritchie D. M., (1988), “The C Programming Language”, 2nd Edition, Prentice Hall
- [12] Lippman S. B., Lajoie J., Moo B. E., (2013), “C++ Primer”, 5th Edition, Objectwrite Inc.
- [13] Web 7, (2019), <https://github.com/osuvak/dev-in-place>, (Date of Access: 22/12/2019)
- [14] Martin D. H., Cordy J. R., (2016), “On the Maintenance Complexity of Makefiles”, 2016 IEEE/ACM 7th International Workshop on Emerging Trends in Software Metrics (WETSoM), 50-56, Austin, TX, USA

- [15] Buffenbarger J., (2013), “Adding Automatic Dependency Processing to Makefile-Based Build Systems with Amake”, 2013 1st International Workshop on Release Engineering (RELENG), 1-4 , San Francisco, CA, USA

BIOGRAPHY

Önder GÖRMEZ was born in Mersin in 1993. In 2015, he successfully completed the Department of Electrical and Electronics Engineering of Faculty of Engineering at Sakarya University. In 2017, he started his graduate study in the Department of Electronic Engineering at Institute of Natural and Applied Sciences of Gebze Technical University. He worked as an embedded software developer in Ges Teknik Inc., which produces a wide variety of building automation controllers, between July 2015 and March 2019. He proud of being part of the Ges Teknik Inc. R&D Team which has developed Turkey's first BACnet compliant and certified building automation control card. He had to leave this beautiful team to work in the aviation industry, which he wanted to be a part of since childhood. Since April 2019, he has been working as an embedded and Linux software engineer at Turkish Technic Inc..

APPENDICES

Appendix A: Publications within the Scope of Thesis

Görmez Ö., Şuvak Ö., (2018), “Shared Object Building Makefiles with Applications for IP Protection”, Gebze Technical University, Institute of Natural and Applied Sciences, Graduate Research Symposium & Presentation Days, 14-16 May 2018.

Görmez Ö., Şuvak Ö., (2019), “Design Details of the Makefile Project and Its Operation”, Gebze Technical University, Institute of Natural and Applied Sciences, Graduate Research Symposium & Presentation Days, 17-18 June 2019.