

Bare Metal Embedded Systems Build Process using GNU Toolchain

In this tutorial, we will see an introduction to the **GNU toolchain** for the embedded system build process. Firstly, we will see an introduction to the GNU toolchain and after that, we will discuss how to use the GNU toolchain to compile the ARM Cortex M4 microcontroller using the Linux command line without using any IDE. In short, this tutorial includes compilation, linking, and uploading code to ARM cortex M4 microcontroller using GNU toolchains. But you can use the same process for other microcontrollers also which GNU toolchains support. In the end, we will see how to transfer the binary generated images from the host machine to the target device.

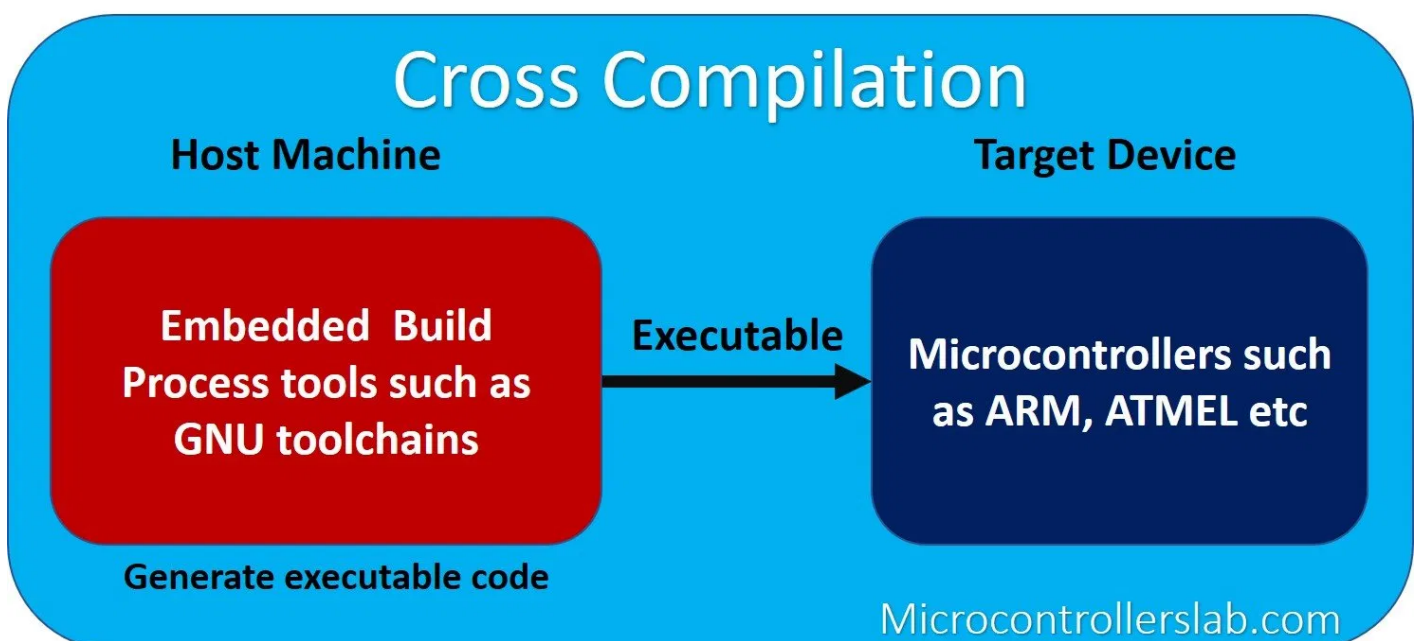
[Table of Contents](#)

Embedded Systems Build Process

Embedded systems build process usually involves a host machine (powerful computer) and resource-constrained **target devices** such as microcontrollers. The tools required to build embedded systems installed on the **host machine**. Because the target device does not have enough resources to install operating systems or embedded built tools such as GNU toolchain. Therefore, we generate a binary image or executable file for the target device using a host machine and compile the application code on a host machine using various tools.

What is Cross Compilation?

The process of generating executable code by using cross toolchains on the hosting machine and created executables runs on the target machine is known as **cross-compilation**. Because we compile executable code on the host machine that runs on the other machine such as target devices i.e. microcontrollers. The main difference between native compilation and cross-compilation is that in a native compilation, the generated executable runs on the same machine.

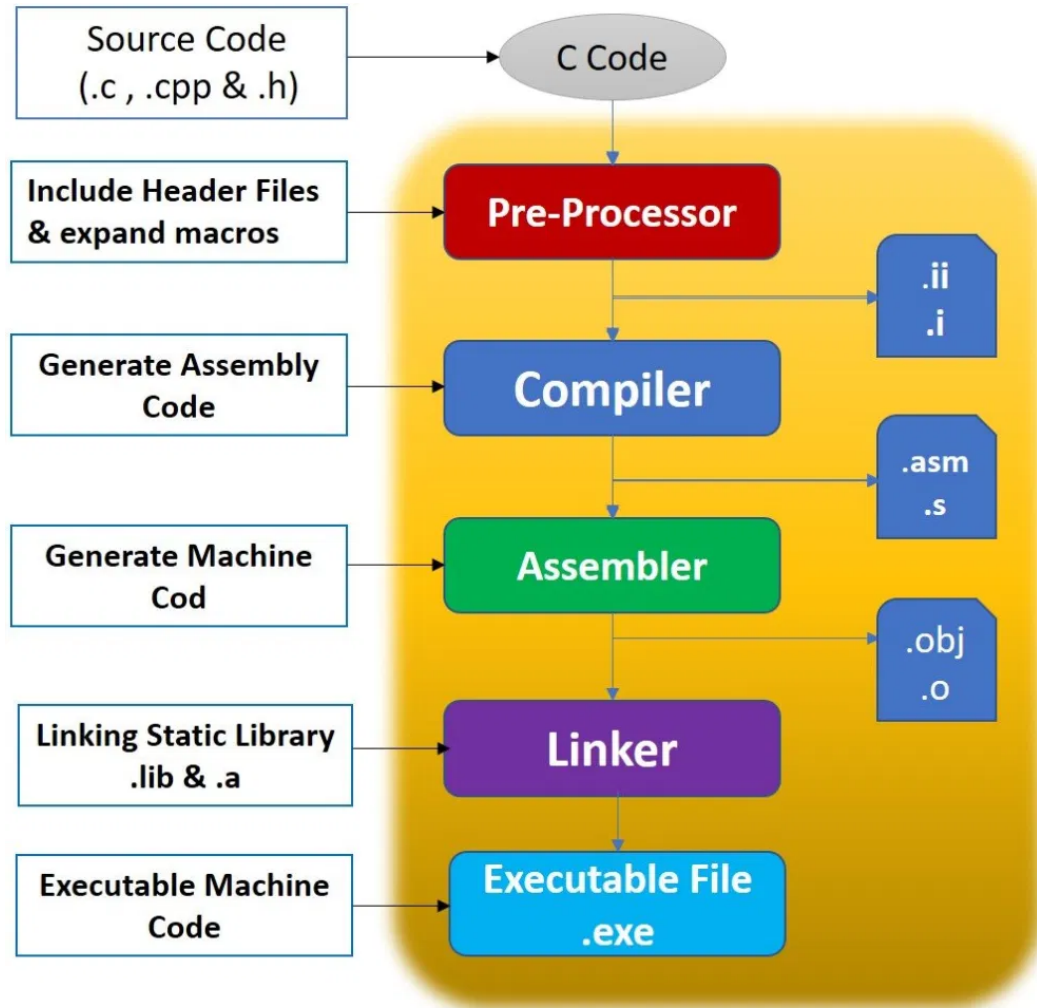


In short, the build process environment for embedded systems runs on the host machine. The host machine generates an executable code that runs on the target embedded device.

GNU toolchains Introduction

GNU toolchain consists of many tools required for embedded system build processes such as preprocessor, compiler, assembler, linker, locator, loader, and debugger. In an embedded system, we usually write embedded applications in c programming language. Your application code goes through various binaries to get an executable file for a target device. GNU cross-compilation toolchain is a collection of these binaries. GNU toolchain also sometimes refers to the GCC toolchain.

Now lets first discuss the function of each [compilation step](#) one by one:



Preprocessing

It performs preprocessing on application source codes (C files and header files) by replacing macros, including header files, and performing conditional compilation.

Compiler

In this stage, the GCC compiler converts the c code into assembly code. The output file extension is a .s file.

Assembler

The output of the compiler is fed into assembler. The assembler converts the assembly code into an object code. The object code is actually a machine code and assembler generates object code according to selected microcontroller or microprocessor instruction set architecture. Your embedded application may have reference to built-in libraries and more

than one source code file. Therefore, the assembler may generate multiple object files. In short, the assembler generates a microcontroller architecture-specific machine code without absolute addresses.

Linker

The assembler generated multiple object files and these object files have references with each other like function calls from other files or use of global variables from other files. Hence, the role of the linker is to combine multiple object files into a single file and also resolve references between these multiple object files using symbols and references table. Linker provides a relocatable file.

Locator

The role of the locator is to perform a memory map according to the selected microcontroller memory map. The locator gets the output file of the linker and maps the code and data into microcontroller memory by providing actual addresses to code and data. It actually defines in which memory region of microcontroller code and data will be saved. Finally, the output of this stage will be ready to use executable code.

Now, we install this executable file on our target device using a loader. Later on in this tutorial, we will see how to use the GNU toolchain loader to upload code to a target device.

Note: C compilation process is architecture-dependent. That means, once you compile source code for a specific microcontroller architecture, it will not run on any other microcontroller having different instruction set architecture.

In summary, the GNU toolchain provides binaries that help to generate and analyze executable files for bare-metal embedded systems:

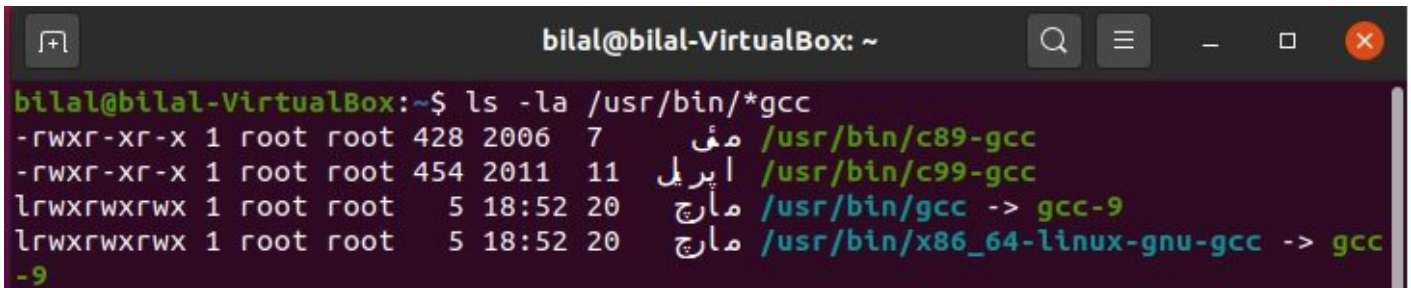
1. Compiling and generating the executable file for a target embedded device
2. Analyzing Memory layout of the generated image
3. Provide many standard C libraries
4. Features to convert an executable file into binary and hex file formats

Install GCC toolchain for ARM Microcontroller

In this section, we will see how to download and install the GCC cross compiler toolchain for ARM microcontrollers. On one Linux based host machine, we can install multiple native or cross-compilers toolchains. GCC toolchains are available in /usr/bin/ directory in Linux. In order to find the list of already installed GCC toolchains on your machine, execute this command on Linux command-line interface.

```
$ ls -la /usr/bin/*gcc
```

On my machine, I got output like of above command like this:



```

bilal@bilal-VirtualBox: ~
bilal@bilal-VirtualBox:~$ ls -la /usr/bin/*gcc
-rwxr-xr-x 1 root root 428 2006 7      مئی /usr/bin/c89-gcc
-rwxr-xr-x 1 root root 454 2011 11     اپریل /usr/bin/c99-gcc
lrwxrwxrwx 1 root root 5 18:52 20     مارچ /usr/bin/gcc -> gcc-9
lrwxrwxrwx 1 root root 5 18:52 20     مارچ /usr/bin/x86_64-linux-gnu-gcc -> gcc-9

```

According to this output, only native gcc compiler is installed on my machine along with C89 and C99 standard libraries.

Now let's see how to install the GCC cross compiler toolchain for bare metal embedded ARM microcontrollers.

First, run this command to update the latest packages and repositories.

```
sudo apt-get update -y
```

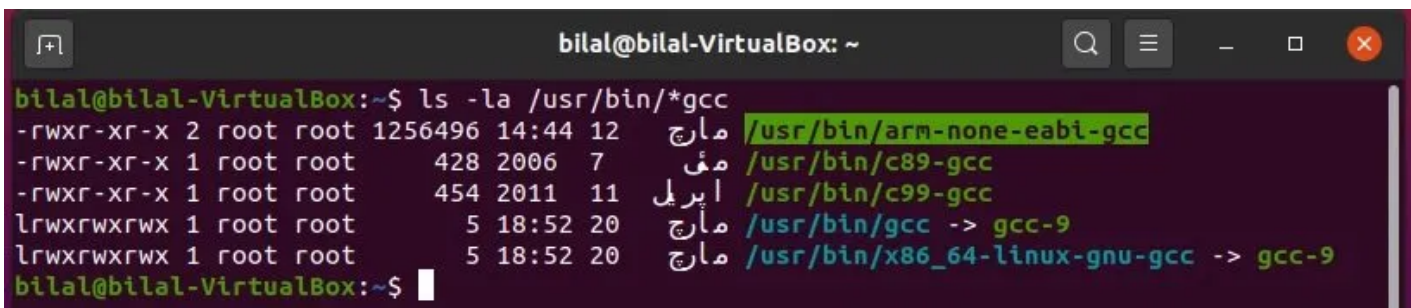
After that run this command to install GCC cross-compilation toolchain for bare metal ARM microcontrollers.

```
sudo apt-get install -y gcc-arm-none-eabi
```

Depending on your system and internet speed, it will take around 5-10 minutes for the installation process to complete. Now again, In order to find the list of installed GCC toolchains on your machine, execute this command on the Linux command-line interface.

```
$ ls -la /usr/bin/*gcc
```

This time you will see that GCC compiler for ARM architecture target devices is also available now in the available list with the name of "**gcc-arm-none-eabi**".



```

bilal@bilal-VirtualBox: ~
bilal@bilal-VirtualBox:~$ ls -la /usr/bin/*gcc
-rwxr-xr-x 2 root root 1256496 14:44 12     مارچ /usr/bin/arm-none-eabi-gcc
-rwxr-xr-x 1 root root 428 2006 7      مئی /usr/bin/c89-gcc
-rwxr-xr-x 1 root root 454 2011 11     اپریل /usr/bin/c99-gcc
lrwxrwxrwx 1 root root 5 18:52 20     مارچ /usr/bin/gcc -> gcc-9
lrwxrwxrwx 1 root root 5 18:52 20     مارچ /usr/bin/x86_64-linux-gnu-gcc -> gcc-9
bilal@bilal-VirtualBox:~$

```

Note: The code compiled with **gcc-arm-none-eabi** cross compiler will not run directly on the host machine. Because executable file generated with ARM cross compiler is architecture-dependent. Similarly, the code compiled with native GCC compiler will also not work on a target devices such as ARM microcontrollers.

Furthermore, you can see all the tools of ARM cross compiler by running this command:

```
ls -la /usr/bin/arm-none-eabi*
```

ARM Cross Compiler Important Commands

These are the important binaries provided by GNU toolchain and they are the most frequently used binaries in the embedded systems building process.

1. **Arm-none-eabi-gcc**: This binary creates an executable file by performing compilation, assembling, and linking steps. Hence, with this one command, we create an executable file for embedded applications.
2. **Arm-none-eabi-as**: This command converts assembly language source code into the architecture-specific machine code.
3. **Arm-none-eabi-ld**: This command is used when we want to use our own linker script. We will talk about the linker script file in coming articles.

These three commands are used to analyze files such as object file, memory map etc.

4. **Arm-none-eabi-objdump**
5. **Arm-none-eabi-readelf**
6. **Arm-none-eabi-nm**

Cross Compiling Code for ARM Microcontroller

Now let's take an example code and see how to cross-compile for ARM-based target devices. The example code is shown below:

```
static int j = 25;
static int i = 10;
int sum = 0;

void main()
{

    sum = i+j;
}
```

This command converts the above-given code into assembly language. This command will just create an assembly file. In other words, only preprocessing and compilation steps will be performed. Here -s will stop the arm GCC after the stage of compilation. As you know the compiler converts c source files into assembly files.

More information on Gcc compiler controlling options are available on this [link](#).

```
arm-none-eabi-gcc -c -mcpu=cortex-m4 -mthumb -S example.c -o example.s
arm-none-eabi-gcc -mcpu=cortex-m4 -mthumb -S example.c -o example.s
```

- -o option places the compiler output file in example.s file
- -mcpu=cortex-m4 specifies the architecture of ARM device
- -mthumb mentions the architecture mode of cortex M4. Because arm cortex m4 microcontroller works in thumb mode only.

The output of the above command generates an assembly file:


```

bilal@bilal-VirtualBox: ~
bilal@bilal-VirtualBox:~$ cat example.c
static int j = 25;
static int i = 10;
int sum = 0;

void main()
{
sum = i+j;
}

bilal@bilal-VirtualBox:~$ arm-none-eabi-gcc -s -mcpu=cortex-m4 -mthumb -S example.c -o example.s
bilal@bilal-VirtualBox:~$ cat example.s
.cpu cortex-m4
.eabi_attribute 20, 1
.eabi_attribute 21, 1
.eabi_attribute 23, 3
.eabi_attribute 24, 1
.eabi_attribute 25, 1
.eabi_attribute 26, 1
.eabi_attribute 30, 6
.eabi_attribute 34, 1
.eabi_attribute 18, 4
.file "example.c"
.text
.data
.align 2
.type j, %object
.size j, 4
j:
.word 25
.align 2
.type i, %object

```

Similarly, by using the proper compiling option, we can create an executable and object files, etc.

Creating and analyzing relocatable object file

In this section, we will see how to create an object file using the ARM GCC cross compiler and analyze the content of the object file using objdump tool of the GNU toolchain.

Object files (.o) have different formats. For example, GCC generates an ELF format that is an executable and linkable format. Other commonly used file formats are COFF, AIF, etc. A file defines the process to store different types of data in different sections of program memory such as .data, .bss, .rodata, and code sections.

For more information on memory organization in a microcontroller, you can read this article:

[Different types of memory in microcontrollers](#)

First, let's create an object file of above given example code by using this command:

```
arm-none-eabi-gcc -c -mcpu=cortex-m4 -mthumb example.c -o example.o
```

This command will save the object file inside an example.o file.

objdump tool

GCC provides arm-none-eabi-objdump binary to analyze various section of the relocatable object file. If you simply run this command in a Linux command line, you will get all options that are available with this command to analyze object file as shown below:

```
arm-none-eabi-objdump
```

```

bilal@bilal-VirtualBox:~$ arm-none-eabi-objdump
Usage: arm-none-eabi-objdump <option(s)> <file(s)>
Display information from object <file(s)>.
At least one of the following switches must be given:
-a, --archive-headers      Display archive header information
-f, --file-headers        Display the contents of the overall file header
-p, --private-headers     Display object format specific file header contents
-P, --private=OPT,OPT...  Display object format specific contents
-h, --[section-]headers   Display the contents of the section headers
-x, --all-headers         Display the contents of all headers
-d, --disassemble         Display assembler contents of executable sections
-D, --disassemble-all    Display assembler contents of all sections
    --disassemble=<sym>   Display assembler contents from <sym>
-S, --source              Intermix source code with disassembly
    --source-comment[=<txt>] Prefix lines of source code with <txt>
-s, --full-contents       Display the full contents of all sections requested
-g, --debugging           Display debug information in object file
-e, --debugging-tags      Display debug information using ctags style
-G, --stabs               Display (in raw form) any STABS info in the file
-W[LLiaprmmFFsoRtUuTgAckK] or
--dwarf[=rawline,=decodedline,=info,=abbrev,=pubnames,=aranges,=macro,=frames,
    =frames-interp,=str,=loc,=Ranges,=pubtypes,

```

For example, -h option displays the different section of memory such as .data, .bss, .rodata, and code sections. Lets analyze the sections of previously created example.o file using this command:

```
arm-none-eabi-objdump -h example.o
```

The output shows the different section:

```

example.o:      file format elf32-littlearm
Sections:
Idx Name          Size      VMA           LMA           File off  Algn
  0 .text          00000028  00000000  00000000  00000034  2**2
    CONTENTS, ALLOC, LOAD, RELOC, READONLY, CODE
  1 .data          00000008  00000000  00000000  0000005c  2**2
    CONTENTS, ALLOC, LOAD, DATA
  2 .bss           00000004  00000000  00000000  00000064  2**2
    ALLOC
  3 .comment       0000005a  00000000  00000000  00000064  2**0
    CONTENTS, READONLY
  4 .ARM.attributes 0000002e  00000000  00000000  000000be  2**0
    CONTENTS, READONLY

```

In the coming tutorials, we will see how to download executable to stm32 using GNU toolchains.

You may also like to read:

- [What is an embedded system? Types and basic building blocks](#)
- [Microcontroller Booting Sequence](#)
- [OVERVIEW OF EMBEDDED SYSTEMS ARCHITECTURE](#)
- [difference between fpga and microprocessor](#)
- [Embedded Systems Applications in the Automobiles industry](#)
- [Difference between stack and heap](#)

