Compilation Process and Memory Allocation

- Compilers are the tool used to translate high-level programming language to low-level programming language.
- Cross Compilers vs. Native Compilers:

1. Native Compiler :

→ Native compilers are compilers that generate code for the same Platform on which it runs.
→ It converts high language into computer's native language.
→ For example, Turbo C or GCC compiler.
→ If a compiler runs on a Windows machine and produces executable code for Windows, then it is a native compiler.
→ The "code generation/compilation" and "running the exactable" happened on the same platform.

2. Cross Compilers:

→ A Cross compiler is a compiler that generates executable code for a platform other than one on which the compiler is running.
→ For example a compiler that running on Windows is building a program which will run on a separate Arduino/ARM.
→ The output executable will run in to "ARM based MCU".
→ Enables developers to compile code for multiple platforms.

Native Compiler	Cross Compiler
Translates program for same hardware/platform/machine on it is running.	Translates program for different hardware/platform/machine other than the platform which it is running.
It is used to build programs for same system/machine & OS it is installed.	It is used to build programs for other system/machine like AVR/ARM.
It can generate executable file like .exe	It can generate raw code .hex
Turbo C or GCC is native Compiler.	Keil is a cross compiler.

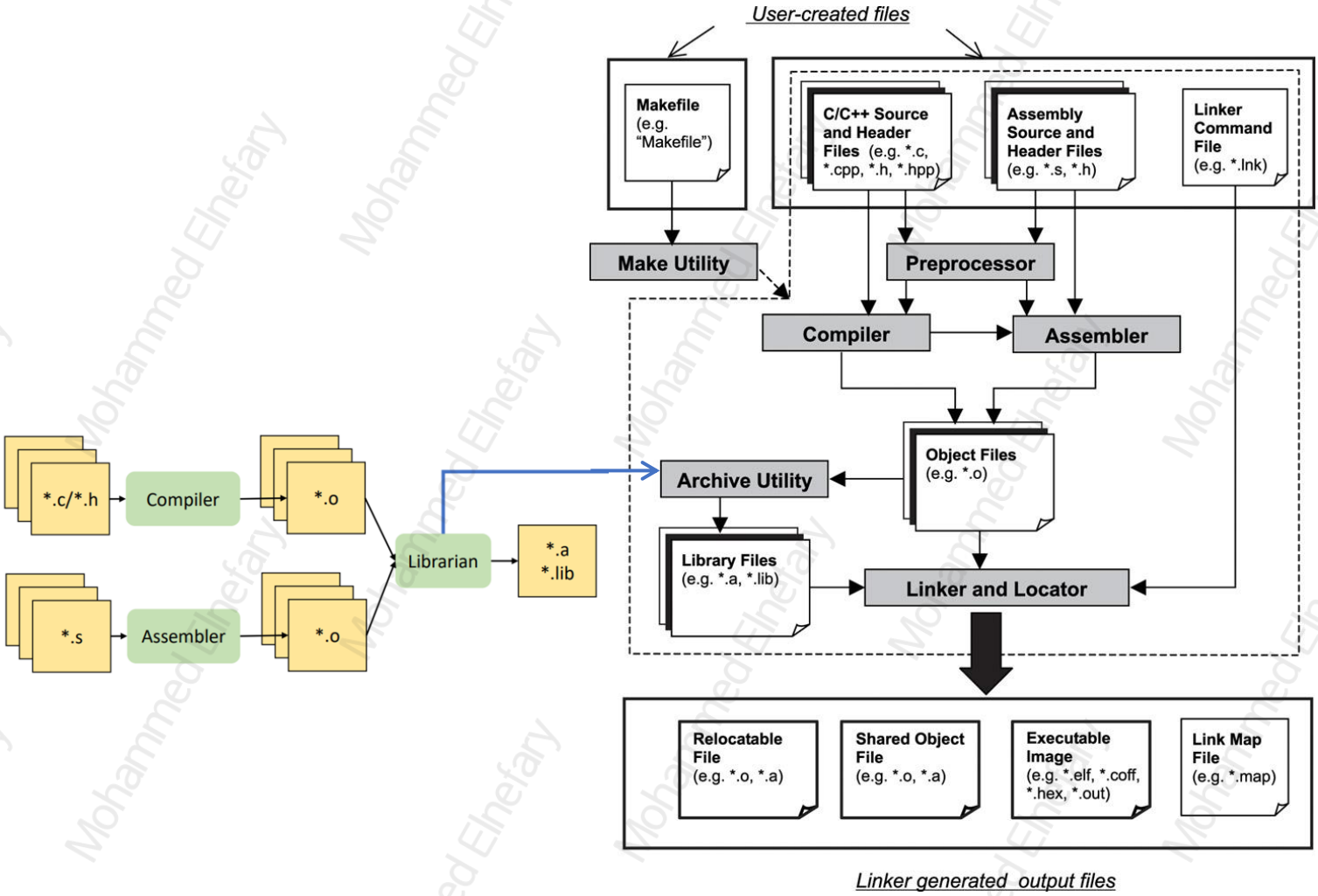
- **The compilation process consists of three main stages:**
  1. **Compilation**
  2. **Linking**
  3. **Loading**

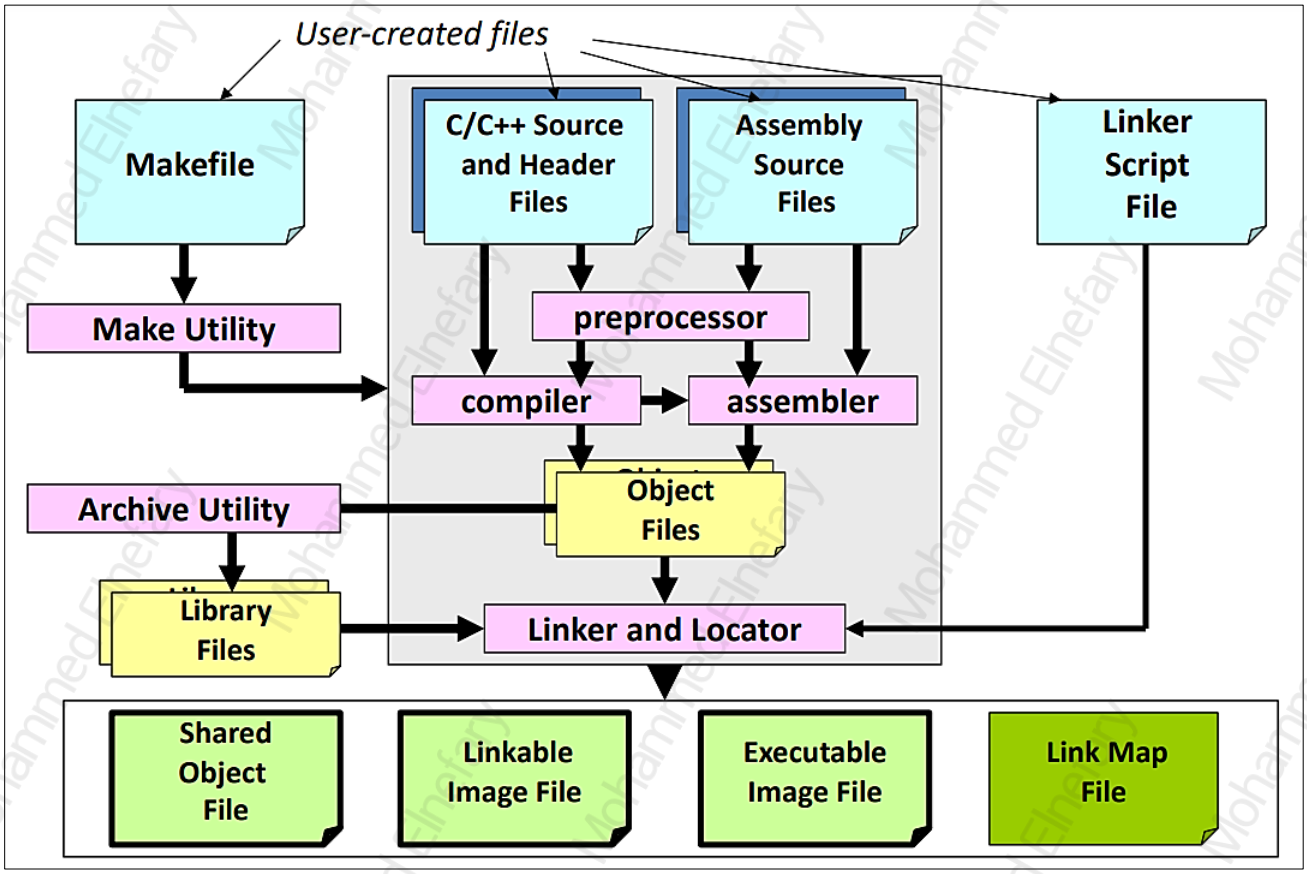
1. Compilation Stage

→ The purpose of this stage is to convert the "**source files**" in to "**Object files / Relocatable Object**".
→ It produces an object file via the assembler
→ The compiler operates on one translation unit at a time.
  - A translation unit typically consists of a single source file (.c file) along with its included header files (.h files).

→ The compiler is responsible for
  - Allocates memory for definitions, including static and automatic variables declared within the source file.
  - Translates the high-level program statements into machine-level opcodes that the processor can execute.

- A Librarian tool may be used to take the object files and combine them into a library file.
- After the compiler converts the source file into an object file, the object file enters an **optional stage**.
  - ✓ It can either undergo processing by a tool called the Archive Utility, also known as the Librarian, which produces files with extensions .a or .lib. Then it enters to the linker stage after.
    → When the file enters the librarian stage, its purpose is to encapsulate and protect the source code from being directly accessible or viewable by others.
  - ✓ Alternatively, it can enter the linker directly to create the executable file and others.
- Developers can utilize the library file (.a /.lib) for their projects but cannot directly access or view the original source code.


User-created files
Makefile (e.g. "Makefile")
C/C++ Source and Header Files (e.g. *.c, *.cpp, *.h, *.hpp)
Assembly Source and Header Files (e.g. *.s, *.h)
Linker Command File (e.g. *.lnk)
Make Utility
Preprocessor
Compiler
Assembler
.c/.h
Compiler
*.o
*.s
Assembler
*.o
Librarian
*.a
*.lib
Archive Utility
Object Files (e.g. *.o)
Library Files (e.g. *.a, *.lib)
Linker and Locator
Relocatable File (e.g. *.o, *.a)
Shared Object File (e.g. *.o, *.a)
Executable Image (e.g. *.elf, *.coff, *.hex, *.out)
Link Map File (e.g. *.map)
Linker generated output files


User-created files
Makefile
C/C++ Source and Header Files
Assembly Source Files
Linker Script File
Make Utility
preprocessor
compiler
assembler
Archive Utility
Object Files
Library Files
Linker and Locator
Shared Object File
Linkable Image File
Executable Image File
Link Map File

- Compilation Stage example (Based On Native Compiler)

You could display the version of GCC via `--version` option:

```
$ gcc --version
gcc (GCC) 6.4.0
```

Help

You can get the help manual via the `--help` option. For example,

```
$ gcc --help
```

To compile the `hello.c`:

```
> gcc hello.c
  // Compile and link source file hello.c into executable a.exe
```

- -Wall : Enable all warning during the compilation process.
- -g : Generate debugging information
- - c : Compile or assemble the source files, but do not link.
- - o : Change the name of the generated object file, if it not used, the default name will be [a.o].

1. Pre-processing: via the GNU C Preprocessor (`cpp.exe`), which includes the headers (`#include`) and expands the macros (`#define`).

```
> cpp hello.c > hello.i          OR      gcc -E hello.c -o hello.i
```

The resultant intermediate file "`hello.i`" contains the expanded source code.

2. Compilation: The compiler compiles the pre-processed source code into assembly code for a specific processor.

```
> gcc -S hello.i
```

The `-S` option specifies to produce assembly code, instead of object code. The resultant assembly file is "`hello.s`".

3. Assembly: The assembler (`as.exe`) converts the assembly code into machine code in the object file "`hello.o`".

```
> as -o hello.o hello.s      Or     gcc  -wall  -g   -c   main.c   -o
```

4. Linker: Finally, the linker (`ld.exe`) links the object code with the library code to produce an executable file "`hello.exe`".

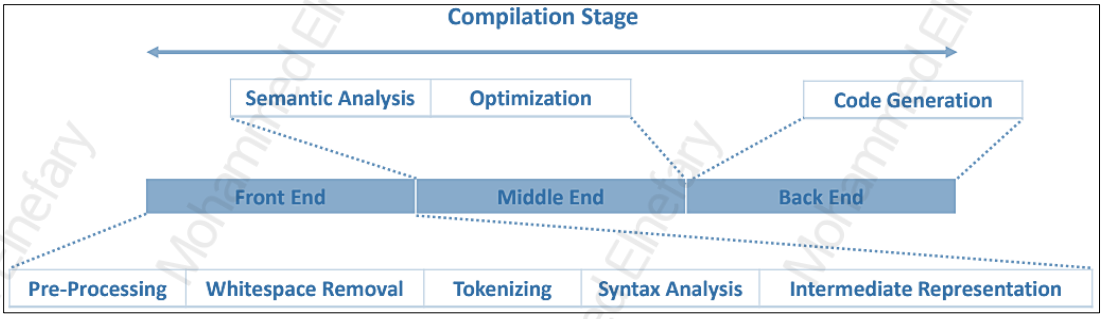
```
> ld -o hello.exe hello.o ...libraries...
```

Verbose Mode (-v)

You can see the detailed compilation process by enabling `-v` (verbose) option. For example,

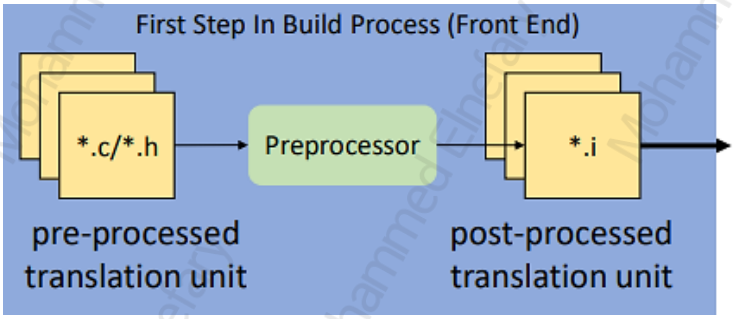
```
> gcc -v -o hello.exe hello.c
```

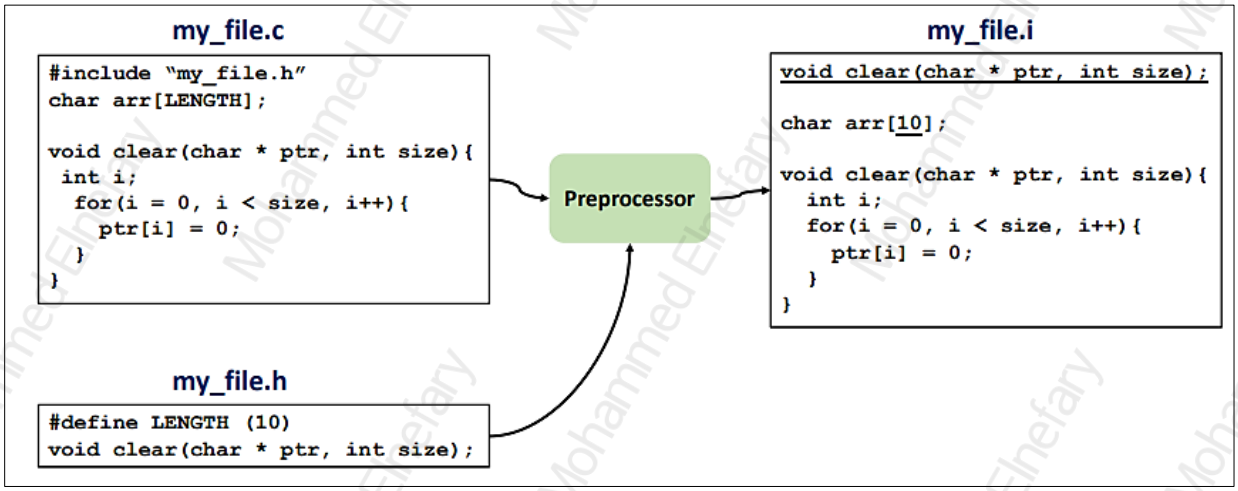
1. **Compilation Stage:** ⟹ **(Compiler + Assembler)**
   - → Compilation stage is a **multi-steps process**.
   - → Each stage working with the output of the previous.
     - The output of the first stage is input to the second stage.
   - → The Compilation Stage itself is normally broken down into three parts:
     1. The "**Front End**": Responsible for parsing the source code.
     2. The "**Middle End**": Responsible for code optimization.
     3. The "**Back End**": Responsible for code generation.



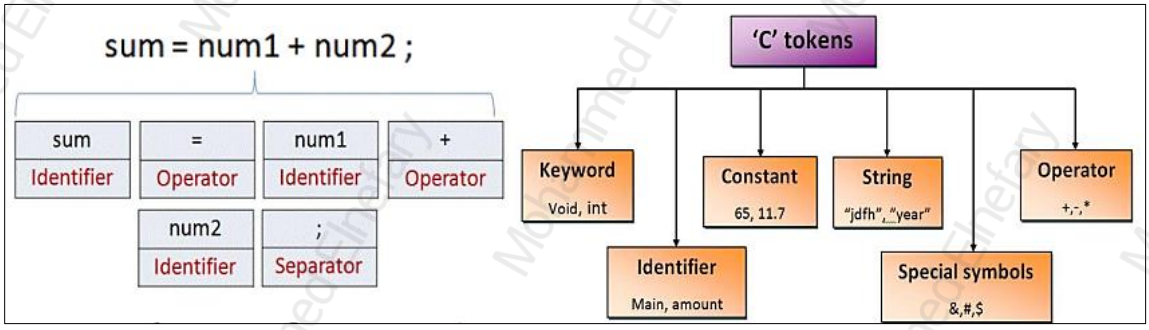
- **The "Front End" steps**
  - → a) Pre-Processing
  - → b) Whitespace Removal
  - → c) Tokenizing
  - → d) Syntax Analysis
  - → e) Intermediate Representation

1. **The "Front End" (Pre-Processing Step): ⟹ (Preprocessor)**
   - → It performs certain source code transformations before the code is processed by the compiler
   - → The preprocessor applies preprocessor directives and macros to a source file, and removes comments.
   - → Substitute macros and inline functions ➔(#define / #if / #ifdef / …….) → text replacement
   - → Expansion of Header files (#include …)
   - → Removes all comments
   - → The preprocessor can be invoked as **gcc -E**.
   - → "-E" ➔ Stop after the preprocessing stage; do not run the compiler proper.
   - → The output is in the form of post processed source code file or "intermediate file"





2. **The "Front End" (Whitespace Removal)**
   - ✓ C ignores whitespace so in the "Whitespace Removal" step, the compiler will strip out all whitespace.
3. **The "Front End" (Tokenizing / Lexical Analysis) ⟹ (Tokenizer)**
   - → A C program is made up of tokens, A token may be:
     - ✓ Keywords ➔ (static-extern-register-auto-const-int……..)
     - ✓ Constants ➔ (constant values → 0, 1, 2, 3…….)
     - ✓ Identifiers ➔ (Variables name)
     - ✓ Symbols ➔ ([ ] - ( ) - { } - ;)
     - ✓ Operators ➔ (== , & , | , < , >……..)
     - ✓ String Literals ➔ (Strings → array of characters)
   - → Ex:
     - ✓ static int x = 4 ;
     - ✓ Read the input characters and produce a sequence of tokens that the parser uses for syntax analysis.
     - ✓ In the compilation process, the tokenizer is responsible for dividing the source code into a set of tokens and ensuring that each token belongs to a valid token category. If a token does not belong to a valid category, it will result in a compilation error.



4. **The "Front End" (Syntax Analysis / Hierarchical Analysis / Parsing) ⟹ (Lexical analyzer / Parser)**
   - → The input to the Lexical Analyzer is a "post-processed" file without spaces and tokenized.
   - → Syntax analysis ensures that tokens are organized in the correct way, according to the rules of the language.
   - → If not, the compiler will produce a **syntax error** at this point.
     - Undeclared variable
     - Missing semicolon
     - Missing brackets and so on
   - → The output of syntax analysis is a data structure known as a **Parse Tree / Syntax Tree**.
   - → Explanation : Syntax Analysis
     - For example: sum = num1 + num2 ;
     - The addition operator plus ('+') in programming languages typically operates on two operands.
     - During the compilation process, the syntax analyzer specifically checks if the plus operator has exactly two operands associated with it.
     - The syntax analyzer does not perform any type checking on the operands accompanying the plus operator.
     - This means that the syntax analyzer does not consider the specific types of the operands when validating the expression.
     - For instance, if one operand is a "string" and the other is an "integer," the syntax analyzer will not raise an error since it solely focuses on the presence of two operands for the plus operator

5. The "Front End" (Intermediate Representation):

→ Not mandatory for all compilers.

→ The output of **syntax analysis** is a data structure known as a **Parse Tree / Syntax Tree**.

→ The **IR (**Intermediate Representation**) program** is generated **from the parse tree**, which comes from Syntax Analysis.

→ The purpose of the IR is to enable the compiler vendor to support multiple programming languages (such as C, C++, etc.) on multiple target platforms without needing to create separate toolchains for each combination.

→ There are several IRs in use, for example Gimple, used by GCC.

→ IRs are typically in the form of an **Abstract Syntax Tree (AST) or Pseudo Code.**

C Code:

```
double sum(double *x, int n) {
  double sum = 0;
  for(int i=0; i<n;++i) {
    sum += x[i];
  }
  return sum;
}
```

Gimple:

```
    assign<sum, constant, 0.0>
    assign<i, constant, 0>
L2: condition<less_than, i, n> L1 else L3
L1: assign<temp1, mult, i, 8>
    assign<temp2, pointer_plus, x, temp1>
    assign<temp3, memref, temp2>
    assign<sum, add, temp3, sum>
    assign<i, add, i, 1>
    goto<L2>
L3: return<sum>
```

➢ The "Middle End" steps

→ Semantic Analysis

→ Optimization

1. Semantic Analysis ⟹ Semantic Analyzer:

→ The Semantic Analyzer examines **the actual meaning** of the statements parsed in the parse tree.

→ Semantic analysis adds further semantic information to the IR AST and performs checks on the logical structure of the program.

→ The type and amount of semantic analysis can vary between different compilers.

→ Modern compilers have the ability to detect potential issues such as unused variables or uninitialized variables during semantic analysis.

→ Semantic analysis can compare information within different parts of the parse tree.

- For example, it can ensure that references to variables align with their declarations or verify that function call parameters match the function's definition.

→ Any problems identified during this stage are typically presented as **warnings** rather than errors.

→ During the "Middle End" stage of the compilation process, the Semantic Analyzer performs various tasks:

- Including the insertion of debug information.
- Constructing of the program symbol table.
  - ✓ The symbol table is a data structure that stores information about scope and details related to names and instances of various entities, such as variable and function names, addresses, and more.

Symbol Table

	Name	Binding	Type	Address	
1	"foo_bar"	Global	Data	0x80	← global variable
2	"do_it"	Global	Function	0x1000	← global function
3	"func_X"	Local	Function	0x2000	← static function
4	"func_a"	Global	Function	UNDEF	← external function

- **Semantic Analyzer** Checking for:
  - ✓ Using of an uninitialized variables
  - ✓ Error in expressions
  - ✓ Array index out of bound
  - ✓ Type compatibility
  - ✓ Using local variable in a different scope
  - ✓ Implicit casting → warning

→ Undeclared variable ➔ parsing ➔ error

→ Using local variable in a different scope ➔ Semantic analysis ➔ error

2. (Optimization) ⟹ (Optimizer):

→ Optimization plays a crucial role in improving the efficiency of the compiled code. By applying various techniques.

→ Optimization techniques are applied to transform the code into a functionally equivalent but smaller or faster form.

→ The compiler aims to generate code that runs faster and uses resources more efficiently.

→ These optimizations can have a significant impact on the execution speed and memory usage of the resulting program

→ Optimization is typically a multi-level process

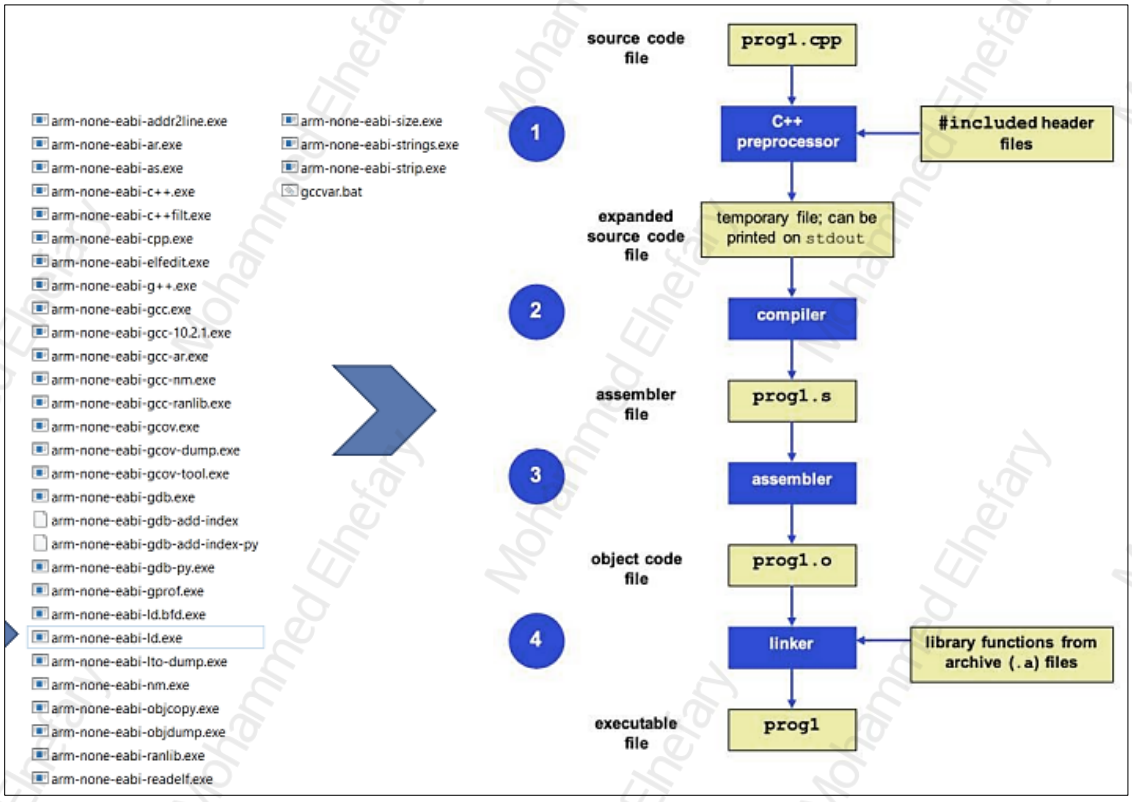
→ There are several common optimization techniques used during this stage, including:

- In-lined expansion of functions: This involves replacing function calls with the actual function code to eliminate the overhead of the function call.
- Dead code removal: Unused or unreachable code is identified and removed from the program, reducing its size and improving execution speed.
- Loop unrolling (loop unwinding): Instead of executing a loop with multiple iterations, the loop body is duplicated to reduce loop overhead and improve execution speed.
- Register allocation: Optimizing the allocation of variables to processor registers to minimize memory accesses and improve performance.
  - ✓ Each register has its own unique address.
  - ✓ The compiler treats the values stored in the registers as constants and doesn't know when they will change.
  - ✓ At that point, the compiler takes one of two actions:
  - ✓ It either copies the data stored in the register and places it in the GPRS (General-Purpose Register Set),
  - ✓ Or it completely removes this place, effectively performing optimization for that location.
- We use volatile keyword to tell the compiler that the value of the variable may change at any time, without any action being taken by the code

- **Some notes about symbol table stages:**
  - → **At Lexical Analysis**
    - During this initial phase, the **Lexical analyzer** scans the source code and identifies tokens (such as keywords, identifiers, literals, and operators).
    - At this stage, the lexer can create entries in the symbol table for tokens encountered. These entries typically include the name of the token and basic information like its type.
    - For example, if the **Lexical analyzer** encounters an identifier like myVar, it can add an entry to the symbol table with the name "myVar" and its type (e.g., integer, float, etc.)
  - → **Syntax Analysis (Parsing):**
    - The parser constructs an Abstract Syntax Tree (AST) from the tokens generated by the lexer.
    - During this phase, additional information is added to the symbol table. Attributes such as scope, dimension, line of reference, and use are associated with identifiers.
    - For instance, the symbol table may record whether an identifier is a local variable, a global variable, or a function parameter.
  - → **Semantic Analysis:**
    - In this phase, the compiler uses the available information in the symbol table to check for semantics.
    - It verifies that expressions and assignments are semantically correct (e.g., type checking) and updates the symbol table accordingly.
    - For example, if you have an expression like a = b + c, the symbol table helps verify that b and c are valid identifiers and have compatible types.
  - → **Intermediate Code Generation:**
    - The symbol table is consulted during intermediate code generation.
    - It provides information about memory allocation (e.g., how much runtime memory is allocated for each variable) and helps manage temporary variables.
    - Temporary variables are often introduced during intermediate code generation to simplify complex expressions.
  - → **Code Optimization:**
    - During optimization, the symbol table assists in machine-dependent optimization.
    - Information from the symbol table can guide the optimization process, improving the efficiency of the generated code.
  - → **Target Code Generation:**
    - Finally, when generating the actual machine code, the symbol table provides the address information for identifiers.
    - The compiler uses this information to generate code that correctly accesses variables and functions.
  - ✓ In summary, the symbol table is created and updated throughout various phases of the compilation process
- **The "Back End" steps:**
  - → Code generation can be considered as the final phase of compilation
    - The compiler produces machine code (object code) from the optimized intermediate representation (IR).
  - → The code generated by the compiler is an object code (Relocatable Object File).
  - → Converts the optimized IR code structure into native opcodes for the target platform.
    - The generated machine code consists of native opcodes specific to the target CPU architecture.
    - These opcodes are directly executable by the processor.
  - → C-code ➔ Assembly code
  - → Logical address distribution

- GCC Compiler for ARM Embedded Processors (GNU Arm Embedded Toolchain)
    - Free and Open source
    - Used to compile (C, C++ and assembly programming).
    - Targets the 32-bit Arm Cortex-A, **Arm Cortex-M**, and Arm Cortex-R processor families.
    - **The main executables are:**
        - arm-none-eabi-gcc.exe: **This is the master driver for the entire toolchain!**
        - arm-none-eabi-as.exe (Assembler)
        - arm-none-eabi-ld.exe (Linker)
    - **This executable used for exchanging the (Format)**
        - arm-none-eabi-objcopy.exe
    - **These are executables used for (.ELF) analyzing:**
        - arm-none-eabi-objdump.exe
        - arm-none-eabi-nm.exe
        - arm-none-eabi-readelf.exe

- arm-none-eabi-gcc.exe
    → It can perform:
    ✓ Preprocessing step ⟶ ✓ Assembling step
    ✓ Compiling step ⟶ ✓ Linking step



- arm-none-eabi-gcc.exe
    → This is the master driver for the entire toolchain.
    → This tool doesn't just compile the code, once compilation is doing it calls the linker which does the linking of separate object files into one big file and locates it by giving proper addresses and produced the final executable.
        - Pre-Processing → Compile → Assemble → Linking → .Object File "Machine Code"

→ Hence gcc can be thought of as a driver program for the entire toolchain as it takes care of the entire process and transforms all the source files of a given project into one final executable.

→ We can make it stop at any point of the entire process using appropriate options as shown below:

```
-E                      Preprocess only; do not compile, assemble or link.
-S                      Compile only; do not assemble or link.
-c                      Compile and assemble, but do not link.
```

- arm-none-eabi-gcc.exe command usage:

`-mthumb`
`-marm`

Select between generating code that executes in ARM and Thumb states. The default for most configurations is to generate code that executes in ARM state, but the default can be changed by configuring GCC with the `--with-mode=`*state* configure option.

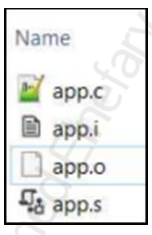
This option specifies the name of the target ARM processor for which GCC should tune the performance of the code. For some ARM implementations better performance can be obtained by using this option. Permissible names are: ‘arm7tdmi’, ‘arm7tdmi-s’, ‘arm710t’, ‘arm720t’, ‘arm740t’, ‘strongarm’, ‘strongarm110’, ‘strongarm1100’, 0‘strongarm1110’, ‘arm8’, ‘arm810’, ‘arm9’, ‘arm9e’, ‘arm920’, ‘arm920t’, ‘arm922t’, ‘arm946e-s’, ‘arm966e-s’, ‘arm968e-s’, ‘arm926ej-s’, ‘arm940t’, ‘arm9tdmi’, ‘arm10tdmi’, ‘arm1020t’, ‘arm1026ej-s’, ‘arm10e’, ‘arm1020e’, ‘arm1022e’, ‘arm1136j-s’, ‘arm1136jf-s’, ‘mpcore’, ‘mpcorenovfp’, ‘arm1156t2-s’, ‘arm1156t2f-s’, ‘arm1176jz-s’, ‘arm1176jzf-s’, ‘generic-armv7-a’, ‘cortex-a5’, ‘cortex-a7’, ‘cortex-a8’, ‘cortex-a9’, ‘cortex-a12’, ‘cortex-a15’, ‘cortex-a17’, ‘cortex-a32’, ‘cortex-a35’, ‘cortex-a53’, ‘cortex-a55’, ‘cortex-a57’, ‘cortex-a72’, ‘cortex-a73’, ‘cortex-a75’, ‘cortex-a76’, ‘cortex-a76ae’, ‘cortex-a77’, ‘cortex-a78’, ‘cortex-a78ae’, ‘cortex-a78c’, ‘ares’, ‘cortex-r4’, ‘cortex-r4f’, ‘cortex-r5’, ‘cortex-r7’, ‘cortex-r8’, ‘cortex-r52’, ‘cortex-m0’, ‘cortex-m0plus’, ‘cortex-m1’, ‘cortex-m3’, ‘cortex-m4’, ‘cortex-m7’, ‘cortex-m23’, ‘cortex-m33’, ‘cortex-m35p’, ‘cortex-m55’, ‘cortex-x1’, ‘cortex-m1.small-multiply’, ‘cortex-m0.small-multiply’, ‘cortex-m0plus.small-multiply’, ‘exynos-m1’, ‘marvell-pj4’, ‘neoverse-n1’, ‘neoverse-n2’, ‘neoverse-v1’, ‘xscale’, ‘iwmmxt’, ‘iwmmxt2’, ‘ep9312’, ‘fa526’, ‘fa626’, ‘fa606te’, ‘fa626te’, ‘fmp626’, ‘fa726te’, ‘xgene1’.

`-mcpu=`*name*`[+extension…]`

This specifies the name of the target ARM processor. GCC uses this name to derive the name of the target ARM architecture (as if specified by `-march`) and the ARM processor type for which to tune for performance (as if specified by `-mtune`). Where this option is used in conjunction with `-march` or `-mtune`, those options take precedence over the appropriate part of this option.

- **arm-none-eabi-gcc --help**

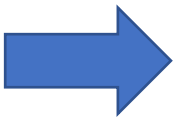
```
Usage: arm-none-eabi-gcc [options] file...
Options:
  -pass-exit-codes         Exit with highest error code from a phase.
  --help                   Display this information.
  --target-help            Display target specific command line options.
  --help={common|optimizers|params|target|warnings|[^]{joined|separate|undocumented}}[,...].
                           Display specific types of command line options.
  (Use '-v --help' to display command line options of sub-processes).
  --version                Display compiler version information.
  -dumpspecs               Display all of the built in spec strings.
```



```
>arm-none-eabi-gcc -E -mcpu=cortex-m4 app.c -o app.i

>arm-none-eabi-gcc -S -mcpu=cortex-m4 app.c -o app.s

>arm-none-eabi-gcc -c -mcpu=cortex-m4 app.c -o app.o
```

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

```c
unsigned short add_one(unsigned short var)
{
    unsigned short l_temp = 0;
    l_temp = var + 1;
    return l_temp;
}
```



```
add_one:
    @ args = 0, pretend = 0, frame = 16
    @ frame_needed = 1, uses_anonymous_args = 0
    @ link register save eliminated.
    push    {r7}
    sub sp, sp, #20
    add r7, sp, #0
    mov r3, r0
    strh    r3, [r7, #6]    @ movhi
    movs    r3, #0
    strh    r3, [r7, #14]   @ movhi
    ldrh    r3, [r7, #6]    @ movhi
    adds    r3, r3, #1
    strh    r3, [r7, #14]   @ movhi
    ldrh    r3, [r7, #14]
    mov r0, r3
    adds    r7, r7, #20
    mov sp, r7
    @ sp needed
    pop {r7}
```

➔ To display the contents of the symbol table:

- **arm-none-eabi-objdump –help ➔ -t**

```
 -t, --syms                Display the contents of the symbol table(s)
```

```
>arm-none-eabi-objdump -t app.o
```

```
app.o:     file format elf32-littlearm

SYMBOL TABLE:
00000000 l    df *ABS*  00000000 app.c
00000000 l    d  .text  00000000 .text
00000000 l    d  .data  00000000 .data
00000000 l    d  .bss   00000000 .bss
00000000 l    d  .rodata        00000000 .rodata
00000000 l    d  .comment       00000000 .comment
00000000 l    d  .ARM.attributes        00000000 .ARM.attributes
00000000 g     O .bss   00000004 NumberOne
00000000 g     O .data  00000002 NumberTwo
00000000 g     O .rodata        00000001 operation
00000000 g     F .text  00000064 main
00000000         *UND*  00000000 printf
00000064 g     F .text  00000020 add_one
00000084 g     F .text  00000018 ChangeAccessLevel_Thread_UnPrivileged
```

- **Compilation Stage output is (Relocatable Object Files): Not ready to be flashed to the MCU.**
- **The output from the compilation stage is the relocatable object files.**
- **The object file contains:**
    - → General information about the object file:
        - File size, data size and source file name.

→ Machine architecture specific binary instructions (Opcodes) and data

→ Symbol table.

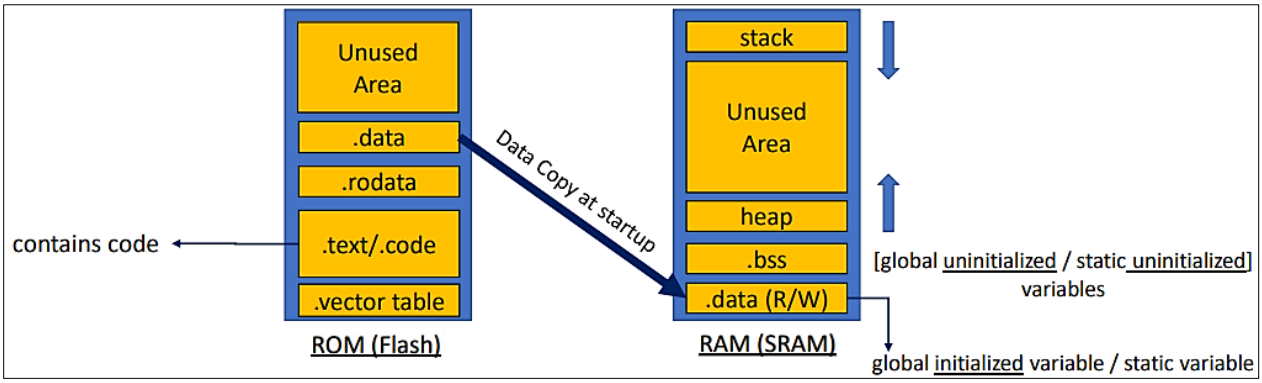
→ Debug information, which the debugger uses

- Normally any source file can contain **CODE and DATA** (Codes operate on data).
- The C compiler allocates memory for code and data in **Sections** and each section contains a different type of information.
- Sections may be identified by name and/or with attributes that identify the type of information contained within this attribute information is used by the Linker for locating sections in memory.

→ Code section will be stored in the code memory (Ex. FLASH memory).

→ Data section will be stored in the code memory (FLASH) or in the data memory (Ex. RAM memory).

- This depends on the nature of the data



→ **.text / .code section** : contains code (Generated opcodes for instructions) ➔ **ROM**

→ **.rodata section** : contains read only data (Global constant) ➔ **ROM**

→ **.data section** : contains any initialized data ➔ **RAM / ROM**

- ✓ Static [global / local] initialized variable / Global initialized variable
- ✓ Each variable has 2 addresses and the **startup code copies the data from LMA to VMA.**

1. **LMA** (Load Memory Address) ➔ **ROM[Flash]**
2. **VMA** (Virtual Memory Address) ➔ **RAM**

→ **.bss section** (Block Started by Symbol) : contains uninitialized ➔ **RAM**

- ✓ Static [global / local] uninitialized variable / Global uninitialized variable
- ✓ A space must be reserved in the RAM by knowing its size with the help of "linker".
- ✓ This space will be initialized to **zeros** with the help of startup code.

→ **.user defined section :**

- ✓ Created by the user, and can contains data or code ➔ **RAM or ROM**
- ✓ Example: Calibration data, Post build Module configurations, code …

→ **.special compiler section :**

- ✓ Add by the compiler. ➔ **ROM**

✓ **Memory Sections:**

→ (**Constants**):

- Constants may come in two forms:
  - → User-defined constant objects (const unsigned int number = 5 ;)
  - → Literals (Floating Point Literals, Integer Literals, Macro Definitions and String Literals)

- Literals are commonly placed in the .text/.code section.
- Most compilers will optimize numeric literals away and use their values directly where possible.
- Many modern C toolchains support a separate **.const/.rodata** section specifically for constant values.
- This section can be placed in (ROM) separate from the .data section.

→ (**Automatic Variables**)

- Variables within functions, including parameters and temporary objects returned from non-void functions, are primarily automatic variables.
- By default, in general programming, memory for these program objects is allocated from the stack.
- Parameters and temporary returned objects have memory allocated by the calling function through pushing values onto the stack.
- Memory allocation for parameters and temporary returned objects occurs when the function is called.
- In this model, automatic memory is reclaimed by popping the stack upon function exit.
- The compiler does not generate a .stack segment; rather, memory allocation takes place within the stack area.
  - → Opcodes are generated to access memory relative to a specific register known as the Stack Pointer.
  - → At program start-up, the Stack Pointer is configured to point to the top of the stack segment.
- **Memory Allocation and Argument Passing:**
  - → The precise memory allocation behavior varies depending on the specific platform.
  - → Example: On x86-64, space for function arguments is not allocated on the stack; instead, they are passed via registers such as %rcx on Windows and %rdi on *nix.
  - → Many RISC architectures like MIPS, PowerPC, and SPARC pass arguments via registers **for simple calls**, rather than using the stack.
  - → ARM Architecture Procedure Call Standard (**AAPCS**) defines which CPU registers are used for function call arguments into, and results from, a function and local variables

→ (**Dynamic Data**)

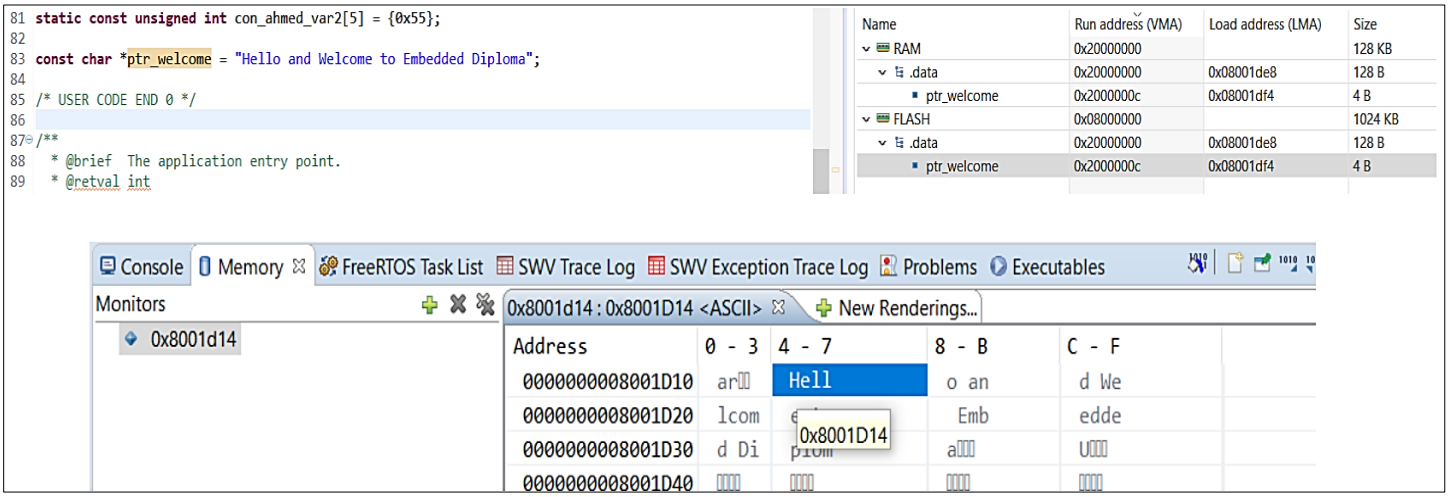
- Memory for dynamic objects is allocated from a section referred to as the Heap.
- The Heap is not allocated by the compiler during compile time; instead, it is allocated by the linker during link-time.
- The sizes of the Heap and Stack are determined in the linker script files.

→ **(Global variable)**

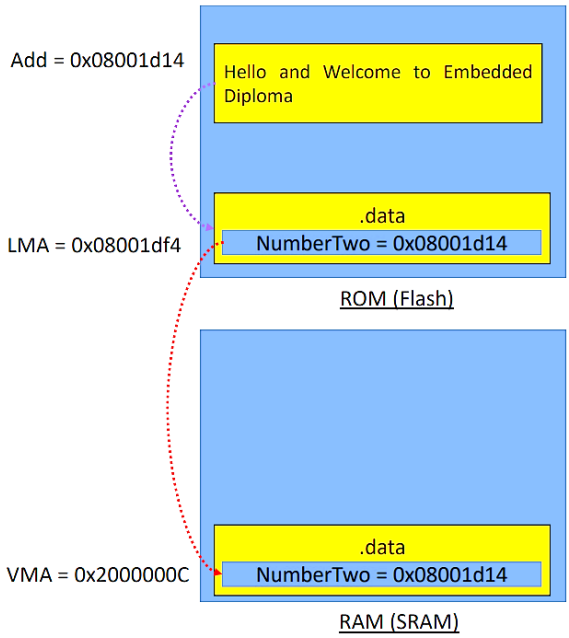
- **Global uninitialized variable**
  - → Allocated to the .bss segment (SRAM) ➔ Value = "0" (Initialized by the startup code).
  - → No LMA, Just VMA.
- **Global initialized variable to value = "0"** ➔ Not Recommended
  - → Allocated to the .bss segment (SRAM) ➔ Value = "0" (Initialized by the startup code).
  - → Compiler can make LMA to these variables which waste the FLASH memory in case of large array size.
- **Global initialized variable**
  - → It will be saved at the .data segment (FLASH) ➔ this means this variable will have "LMA".
  - → It will be allocated to the .data segment (SRAM) ➔ this means this variable will have "VMA".
  - → It is the startup code to copy the value from the (LMA) to the (VMA).

- **Global static uninitialized variable**
  - → Allocated to the .bss segment (SRAM) ➔ Value = “0” (Initialized by the startup code).
- **Global static initialized variable** ➔ can’t be shared “extern” to another source file.
  - → It will be saved at the .data segment (FLASH) ➔ this means this variable will have “LMA”.
  - → It will be allocated to the .data segment (SRAM) ➔ this means this variable will have “VMA”.
  - → It is the startup code to copy the value from the (LMA) to the (VMA).
- **Global constant data & Private Global constant data**
  - → These 2 constants (Private and Shared) data will be allocated to the .rodata (FLASH) ➔ Only Readable

→ **Global pointer, point to a literal string**



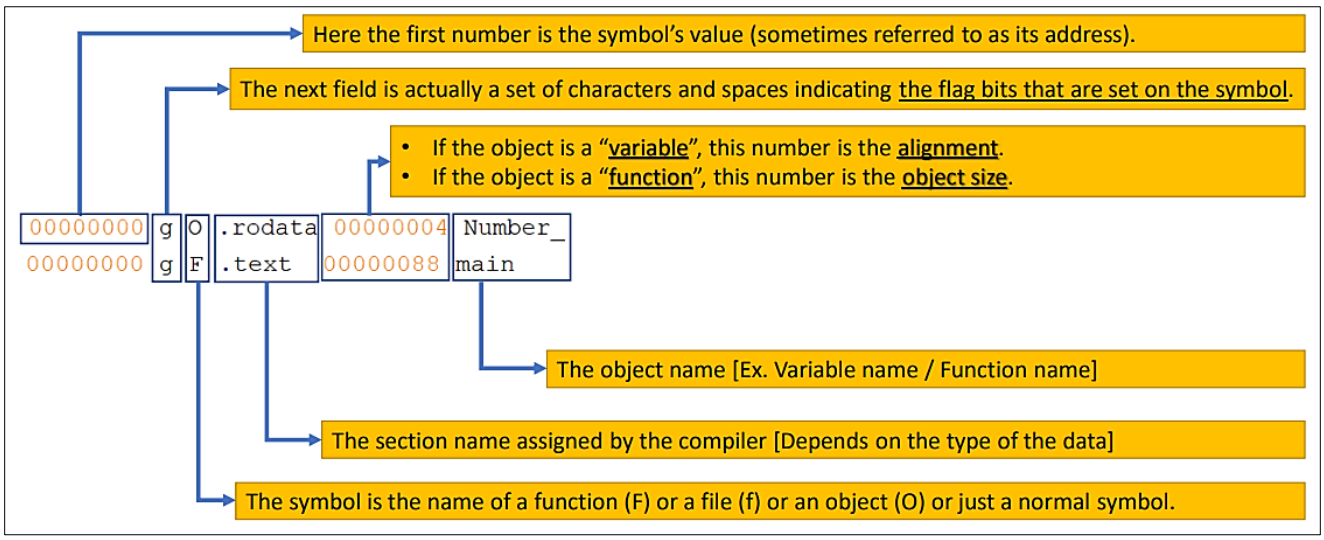
- Ptr_welcome is a global pointer points to a constant character data ‘H’.
  - → This means this variable will need “LMA”.
- The "Hello and Welcome to Embedded Diploma “will be stored at the (FLASH).
- Since the (ptr_welcome) is a global pointer and initialized, so It will be allocated to the .data segment (SRAM)
  - → This means this variable will have “VMA”.



Variable (Data)	LOAD time	RUN time	Section	Note
Global initialized	FLASH	RAM	.data	Should be copied from flash to ram by startup code
Global uninitialized	----	RAM	.bss	startup code reserves space for this data in RAM and initializes to zero
Global static initialized	FLASH	RAM	.data	Should be copied from flash to ram by startup code
Global static uninitialized	----	RAM	.bss	startup code reserves space for this data in RAM and initializes to zero
Local initialized	----	STACK(RAM)	----	Consumed at run time
Local uninitialized	----	STACK(RAM)	----	Consumed at run time
Local static initialized	FLASH	RAM	.data	Should be copied from flash to ram by startup code
Local static uninitialized	----	RAM	.bss	startup code reserves space for this data in RAM and initializes to zero
All global const	FLASH	----	.rodata	
All local const	----	STACK(RAM)	----	Treated as locals

→ **Object File (Symbol Table):**

- ✓ Symbol table is an important data structure created and maintained by compilers in order to store information about entities such as variable names, function names, objects, classes, interfaces, etc.
  - Contains any extern identifiers defined within this translation unit ➔ Exports.
  - Contains any identifiers declared and used within the translation, but not defined within it ➔ Imports.
- ✓ Symbol table analysis using arm-none-eabi-objdump:
  - arm-none-eabi-objdump -t



The flag characters are divided into 7 groups as follows:

l
g
u
!

The symbol is a local (l), global (g), unique global (u), neither global nor local (a space) or both global and local (!). A symbol can be neither local or global for a variety of reasons, e.g., because it is used for debugging, but it is probably an indication of a bug if it is ever both local and global. Unique global symbols are a GNU extension to the standard set of ELF symbol bindings. For such a symbol the dynamic linker will make sure that in the entire process there is just one symbol with this name and type in use.

w

The symbol is weak (w) or strong (a space).

C

The symbol denotes a constructor (C) or an ordinary symbol (a space).

W

The symbol is a warning (W) or a normal symbol (a space). A warning symbol's name is a message to be displayed if the symbol following the warning symbol is ever referenced.

I
i

The symbol is an indirect reference to another symbol (I), a function to be evaluated during reloc processing (i) or a normal symbol (a space).

d
D

The symbol is a debugging symbol (d) or a dynamic symbol (D) or a normal symbol (a space).

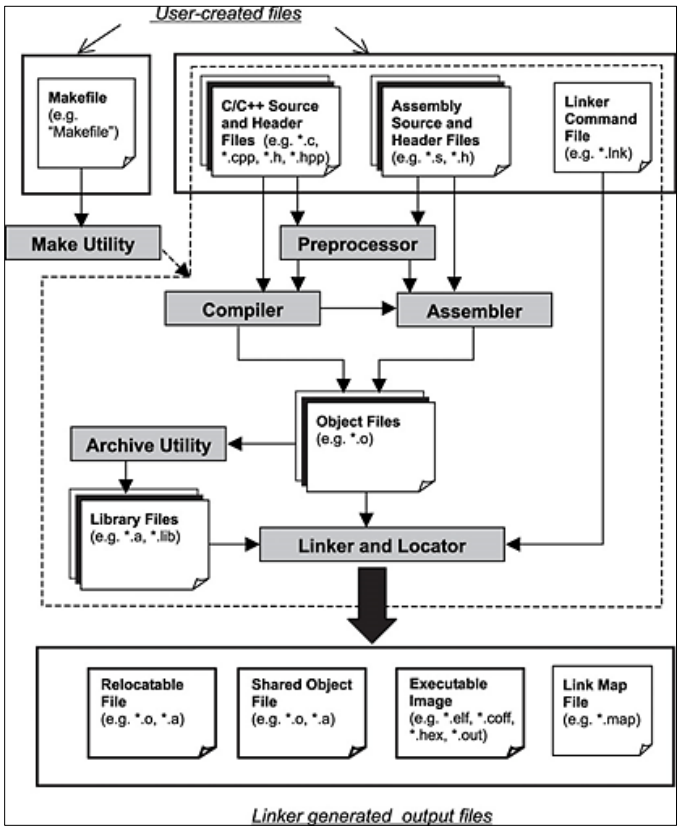
F
f
O

The symbol is the name of a function (F) or a file (f) or an object (O) or just a normal symbol (a space).

```
app.o:     file format elf32-littlearm

SYMBOL TABLE:
00000000 l    df *ABS*  00000000 app.c
00000000 l    d  .text  00000000 .text
00000000 l    d  .data  00000000 .data
00000000 l    d  .bss   00000000 .bss
00000000 l    d  .rodata        00000000 .rodata
00000000 l    d  .comment       00000000 .comment
00000000 l    d  .ARM.attributes        00000000 .ARM.attributes
00000000 g     O .bss   00000004 NumberOne
00000004 g     O .bss   00000004 NumberOne_1
00000008 g     O .bss   00000004 NumberOne_2
0000000c g     O .bss   00000004 NumberOne_3
00000000 g     O .data  00000004 NumberTwo
00000004 g     O .data  00000004 NumberTwo_1
00000008 g     O .data  00000004 NumberTwo_2
00000000 g     F .text  00000016 AddOne
00000016 g     F .text  00000016 AddTwo
0000002c g     F .text  0000009c main
00000000         *UND*  00000000 printf
```

```
unsigned int NumberOne;
unsigned int NumberOne_1;
unsigned int NumberOne_2;
unsigned int NumberOne_3;
unsigned int NumberTwo = 3;
unsigned int NumberTwo_1 = 3;
unsigned int NumberTwo_2 = 3;

unsigned int AddOne(unsigned int Number){
    return Number + 1;
}
unsigned int AddTwo(unsigned int Number){
    return Number + 2;
}

int main()
{
```

2. Linking Stage: ⟹ (Linker and Locator)



→ The Linker **combines** the object files into a single executable program.

→ **The inputs to the linker:**

- Linker command file (Linker Script File)
- Relocatable object files
- Library files ➔ (Output from the "Archive Utility" OR C standard library)

→ **The output from the linker:**

- ✓ Relocatable file
- ✓ Shared Object
- ✓ Executable image
- ✓ Map file

→ **Linking Stage** : Mapping Executable Images into Target Embedded Systems

→ The linker combines input sections having the same name into a single output section with that name by default.

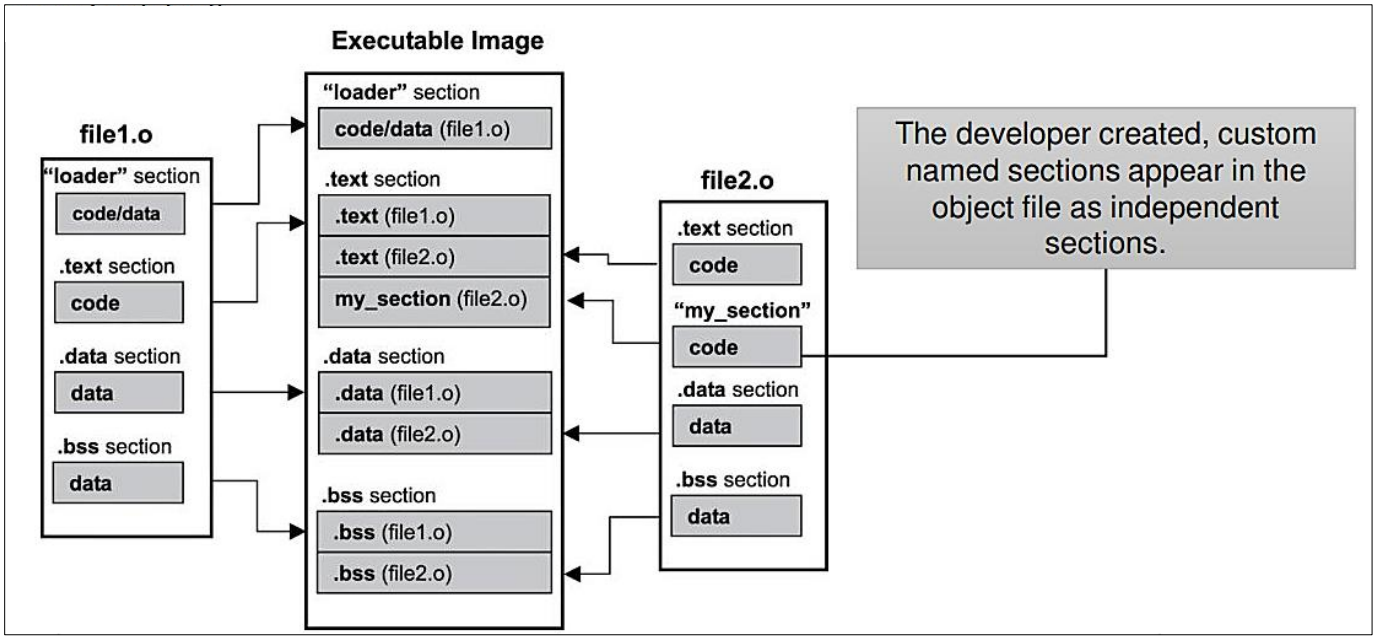
→ When creating relocatable output, the compiler generates the address that, for each symbol, is relative to the file being compiled ➔ these addresses called "**Relative Addressed**".

→ These addresses are generated with respect to **offset 0.**

→ Each relocatable object module "m", has a symbol table that contains info about the symbols that are defined and referenced by "m", There are three different kinds of symbols: Global, External and Local

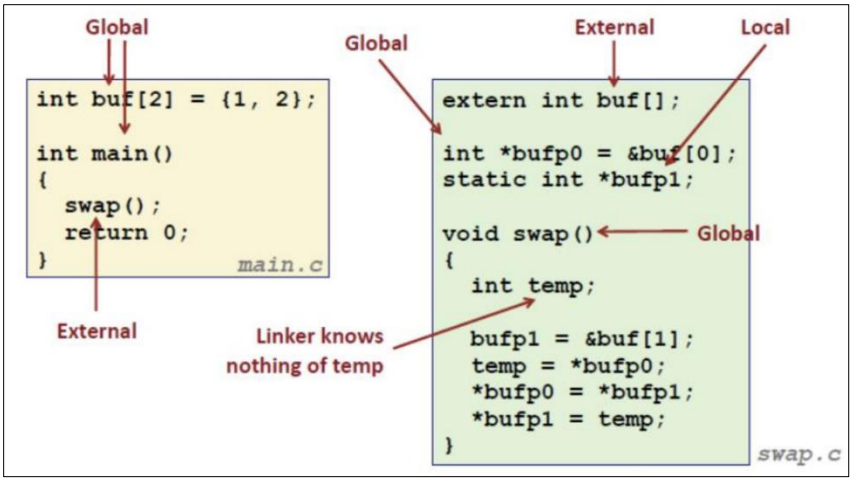
- **Global**: global symbols that are defined by module "m" and that can be referenced by other modules.
  - ✓ Global linker symbols correspond to non-static functions and global variables that are defined without the static attribute.
- **External**: global symbols that are referenced by module "m" but defined by some other module.
  - ✓ Such symbols are called externals and correspond to functions and variables that are defined in other modules.

- **Local (static):** local symbols those are defined and referenced exclusively by module "m".
  - ✓ Some local linker symbols correspond-functions and global variables that are defined with the static attribute.
  - ✓ These symbols are visible anywhere within module "m", but cannot be referenced by other modules.

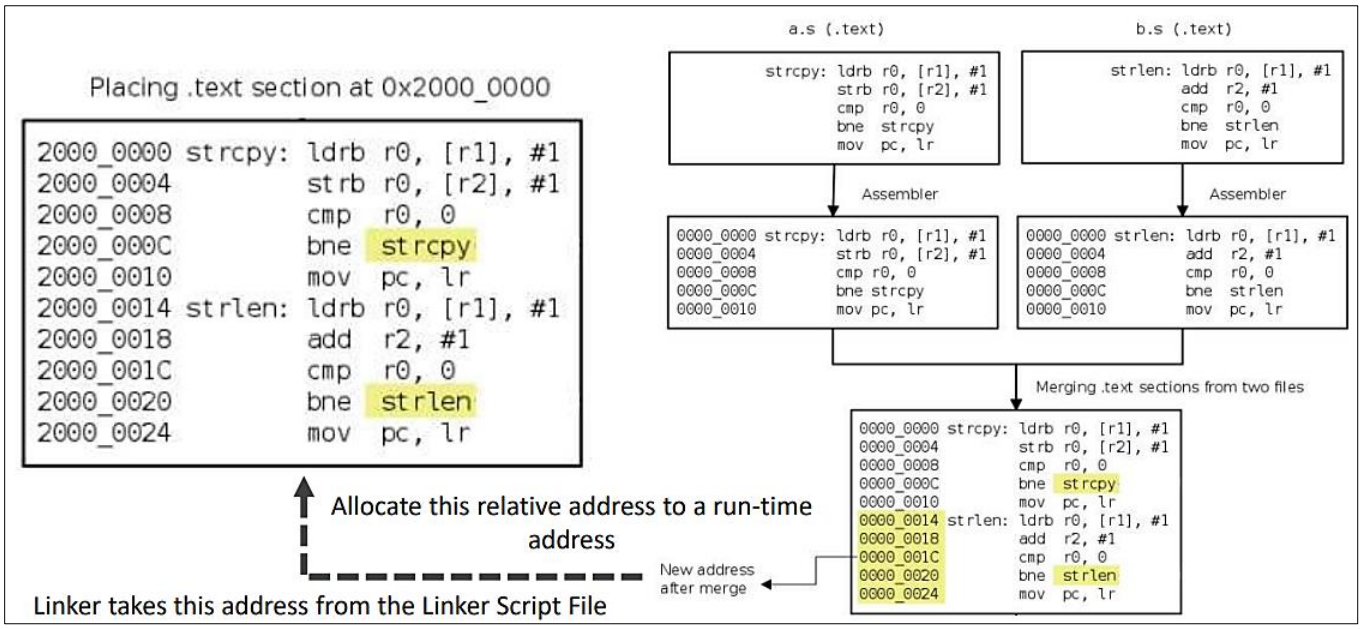



```
app.o:     file format elf32-littlearm

SYMBOL TABLE:
00000000 l    df *ABS*  00000000 app.c
00000000 l    d  .text  00000000 .text
00000000 l    d  .data  00000000 .data
00000000 l    d  .bss   00000000 .bss
00000000 l    d  .rodata        00000000 .rodata
00000000 l    d  .comment       00000000 .comment
00000000 l    d  .ARM.attributes        00000000 .ARM.attributes
00000000 g     O .bss   00000004 NumberOne
00000004 g     O .bss   00000004 NumberOne_1
00000008 g     O .bss   00000004 NumberOne_2
0000000c g     O .bss   00000004 NumberOne_3
00000000 g     O .data  00000004 NumberTwo
00000004 g     O .data  00000004 NumberTwo_1
00000008 g     O .data  00000004 NumberTwo_2
00000000 g     F .text  00000016 AddOne
00000016 g     F .text  00000016 AddTwo
0000002c g     F .text  0000009c main
00000000         *UND*  00000000 printf
```

```c
unsigned int NumberOne;
unsigned int NumberOne_1;
unsigned int NumberOne_2;
unsigned int NumberOne_3;
unsigned int NumberTwo = 3;
unsigned int NumberTwo_1 = 3;
unsigned int NumberTwo_2 = 3;

unsigned int AddOne(unsigned int Number){
    return Number + 1;
}
unsigned int AddTwo(unsigned int Number){
    return Number + 2;
}

int main()
{
```



➢ **Linking Stage :** <u>combines input sections having the same name into a single output section</u>



➢ **Linking Stage :**

→ In a typical program, a section of code in one source file can reference variables defined in another source file.

→ A function in one source file can call a function in another source file.

→ The global variables and non-static functions are commonly referred to as global symbols.

→ The compiler creates a symbol table containing the symbol name to address mappings as part of the object file it produces.

→ The symbol table contains the global symbols defined in the file being compiled, as well as the external symbols referenced in the file that the linker needs to resolve.

→ **The linking process performed by the linker involves:**

- **Symbol resolution**
- **Symbol relocation**

1. Symbol resolution: ➔ This is an "Iterative Process".

→ The process in which the linker goes through each object file and determines, for the object file, in which (other) object file or files the external symbols are defined.

→ Sometimes the linker must process the list of object files multiple times while trying to resolve all of the external symbols.

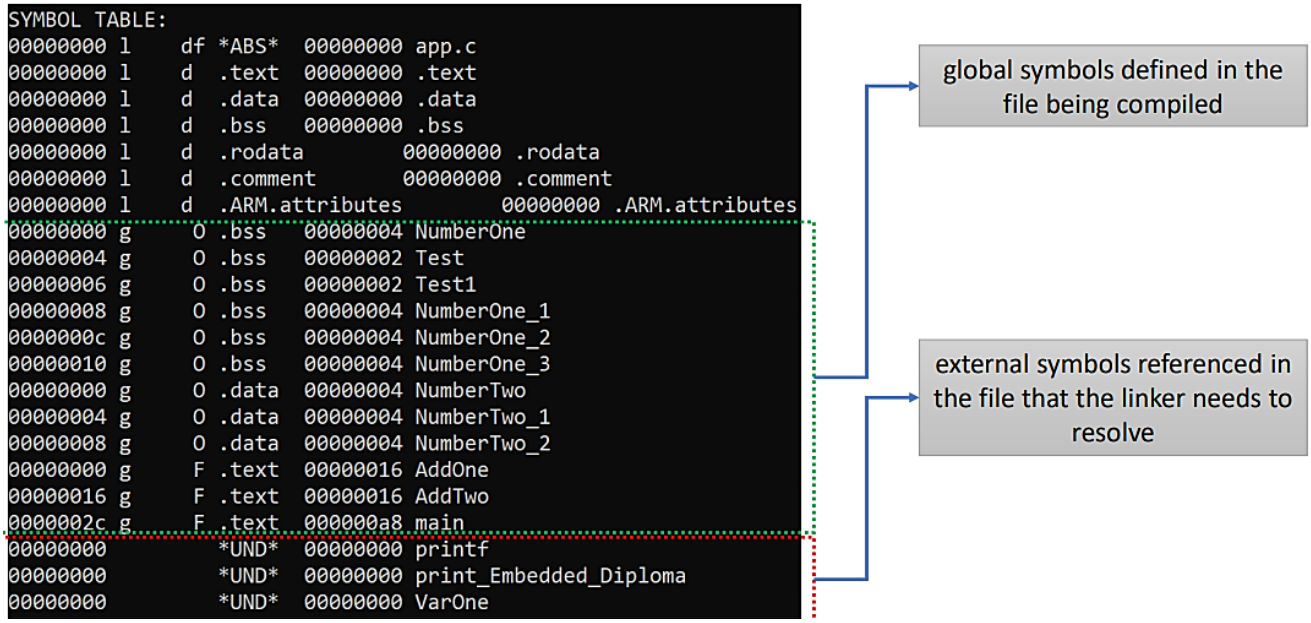
→ When external symbols are defined in a static library, the linker copies the object files from the library and writes them into the final image (Static Linking).

- **Static linking** includes the object code from external libraries directly into the final executable
- While **dynamic linking** includes <u>information (addresses)</u> about the libraries and loads them at runtime.

→ The linker ensures each symbol defined by the program has a unique address.

→ If, after this, the Linker still cannot resolve a symbol it will report an 'Unresolved Reference' error

→ If the Linker finds the same symbol defined in two object files it will report a 'Redefinition' error.



- **Linking Stage :** combines input sections having the same name into a single output section

```
>arm-none-eabi-gcc -c -mcpu=cortex-m4 -mthumb file1.c -o file1.o
>arm-none-eabi-objdump -t file1.o
```

```
>arm-none-eabi-gcc -c -mcpu=cortex-m4 -mthumb file2.c -o file2.o
>arm-none-eabi-objdump -t file2.o
```

```
>arm-none-eabi-ld file1.o file2.o -o app.o
```

```
file1.o:     file format elf32-littlearm

SYMBOL TABLE:
00000000 l    df *ABS*  00000000 file1.c
00000000 l    d  .text  00000000 .text
00000000 l    d  .data  00000000 .data
00000000 l    d  .bss   00000000 .bss
00000000 l    d  .comment       00000000 .comment
00000000 l    d  .ARM.attributes        00000000 .ARM.attributes
00000000 g     O .data  00000004 var1
00000004 g     O .data  00000004 var2
00000000 g     O .bss   00000004 var3
00000004 g     O .bss   00000004 var4
00000000 g     F .text  00000044 fun1
00000000         *UND*  00000000 fun2
```

```
file2.o:     file format elf32-littlearm

SYMBOL TABLE:
00000000 l    df *ABS*  00000000 file2.c
00000000 l    d  .text  00000000 .text
00000000 l    d  .data  00000000 .data
00000000 l    d  .bss   00000000 .bss
00000000 l    d  .comment       00000000 .comment
00000000 l    d  .ARM.attributes        00000000 .ARM.attributes
00000000 g     O .data  00000004 var5
00000004 g     O .data  00000004 var6
00000000 g     O .bss   00000004 var7
00000004 g     O .bss   00000004 var8
00000000 g     F .text  00000044 fun2
```

Sections Merge
Sections Relocation

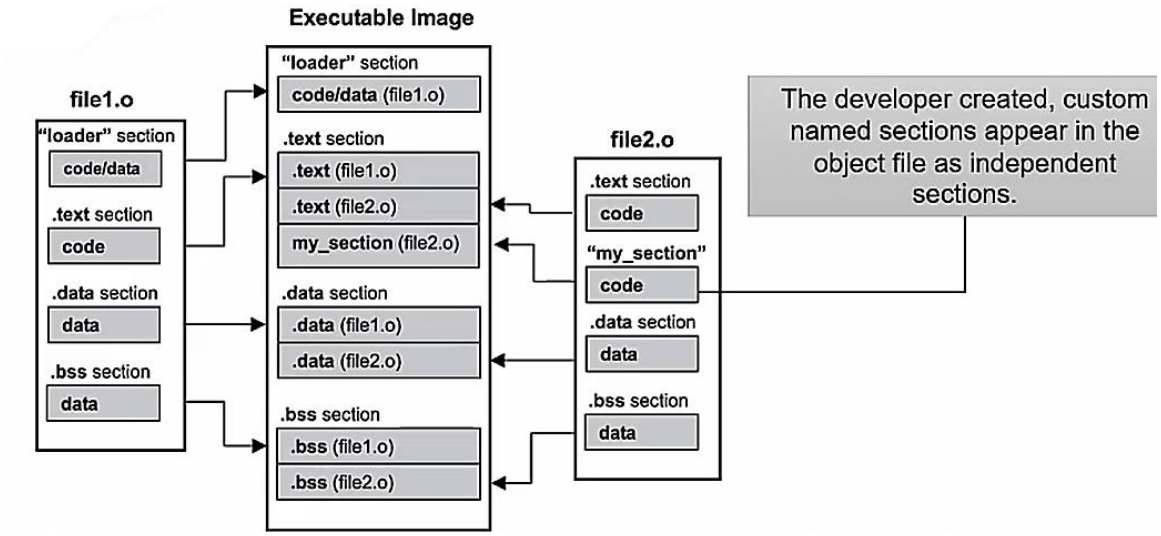
```
app.o:     file format elf32-littlearm

SYMBOL TABLE:
00008000 l    d  .text  00000000 .text
00018088 l    d  .data  00000000 .data
00018098 l    d  .bss   00000000 .bss
00000000 l    d  .comment       00000000 .comment
00000000 l    d  .ARM.attributes        00000000 .ARM.attributes
000180a8 l    d  .noinit        00000000 .noinit
00000000 l    df *ABS*  00000000 file1.c
00000000 l    df *ABS*  00000000 file2.c
00018090 g     O .data  00000004 var5
000180a8 g       .bss   00000000 _bss_end__
00018098 g       .bss   00000000 __bss_start__
00018088 g     O .data  00000004 var1
0001809c g     O .bss   00000004 var4
000180a4 g     O .bss   00000004 var8
000180a8 g       .bss   00000000 __bss_end__
00000000         *UND*  00000000 _start
00008044 g     F .text  00000044 fun2
00018098 g       .bss   00000000 __bss_start
0001808c g     O .data  00000004 var2
000180a8 g       .bss   00000000 __end__
00018094 g     O .data  00000004 var6
00018098 g     O .bss   00000004 var3
000180a0 g     O .bss   00000004 var7
00018098 g       .data  00000000 _edata
000180a8 g       .bss   00000000 _end
00008000 g     F .text  00000044 fun1
00080000 g       .comment       00000000 _stack
00018088 g       .data  00000000 __data_start
```

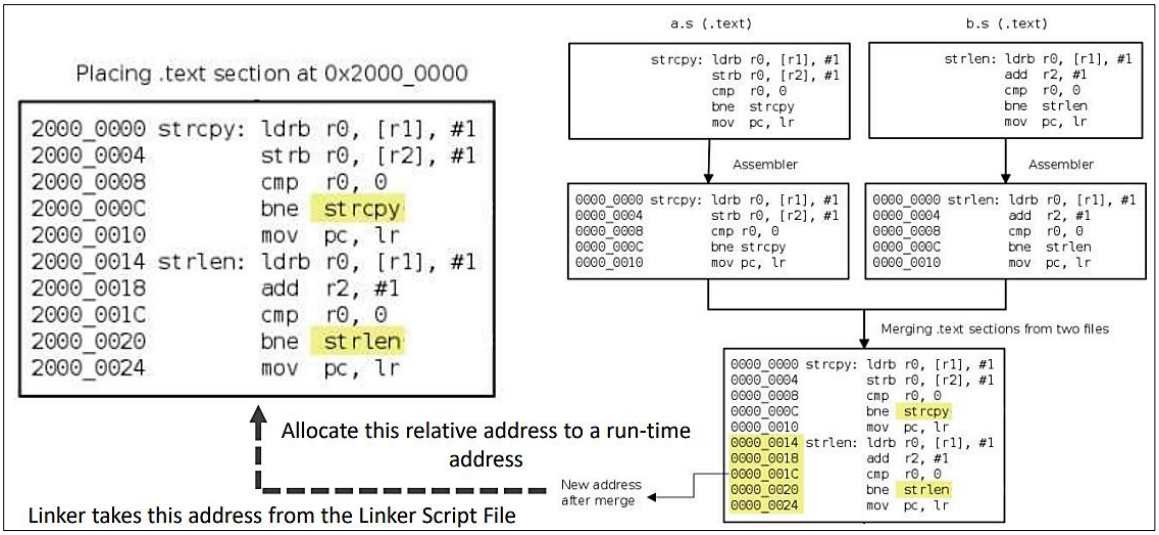
2. Symbol Relocation / Section Location / Locating ➔ Locator:

1. Once the linker has completed the symbol resolution step, it has associated each symbol reference in the code with exactly one symbol definition.
2. At this point, the linker knows the exact sizes of the code and data sections in its input object modules.
3. The relocation step begins, merging input modules and assigning runtime addresses to symbols.
4. Code and data sections must have absolute addresses in memory for execution.
5. Each section is assigned a specific absolute address in memory.
6. This can be done on a section-by-section basis but more commonly sections are concatenated from some base address
7. Relocation includes two steps:
    a) Relocating sections and symbol definitions.
        - Sections of the same type are merged into a new aggregate section.
        - The linker then assigns run-time memory address to the new aggregate sections, to each section defined by the input modules, and to each symbol defined by the input modules.
        - When this step is complete, every global symbol in the program has a unique run-time memory address.
    b) Relocating symbol references within sections
        - In this step, the linker modifies every symbol reference in the bodies of the code and data sections so that they point to the correct run-time address.
        - To perform this step, the linker relies on data structures in the relocatable object modules known as relocation entries.

→ During the relocation process, the linker ensures that symbol references and definitions are correctly associated and that code and data sections are assigned appropriate runtime memory addresses. This allows the program to be executed successfully, as each symbol reference points to the correct runtime address. The linker merges sections, assigns addresses, and modifies references as necessary, utilizing relocation entries within the relocatable object modules.

→ Symbol relocation requires code modification because the linker adjusts the machine code referencing these symbols to reflect their finalized addresses.





→ The linker's task is to combine object files and merge sections from different object files into program segments, resulting in a single executable image for the target embedded system.

→ To achieve this, embedded developers utilize linker commands, also known as linker directives, to control the linker's behavior when combining sections and allocating segments in the target system. These linker directives are stored in a **linker command/script file.**

→ The primary objective of creating a linker script file is to accurately and efficiently map the executable image into the target system.

→ *Linker script files provide instructions to the linker regarding **how to combine object files** and **where to place the binary code and data in the memory of the embedded system.***

→ The format of linker command files and the syntax of linker directives can vary between different linkers.

- It is advisable to consult the reference manual of the specific linker being used to understand the available commands, syntaxes, and extensions for that particular linker.

→ However, there are some commonly used linker directives that are shared among many linkers. Examples of such directives include

- "MEMORY," which specifies memory regions and their attributes
- And "SECTION," which controls the placement of specific sections in memory.

→ By utilizing the MEMORY and SECTION directives in the linker command file, an embedded developer can accurately control the memory mapping and section merging process, ensuring that the code and data are placed correctly in the target system's memory.

→ Using the MEMORY directive, the linker can describe the memory map of the target system.

→ The memory map lists different types of memory (e.g., RAM, ROM, flash) on the target system, each with a specific range of addresses.

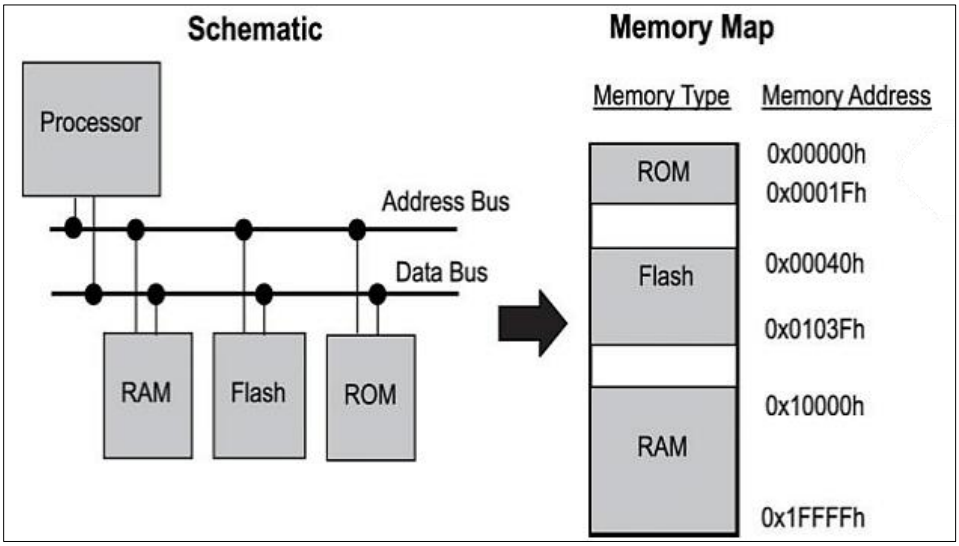
→ Before creating a linker command file, an embedded developer needs to have a good understanding of the addressable physical memory on the target system.

→ The MEMORY directive is used to define the types of physical memory present on the target system and the address range occupied by each physical memory block, which includes three physical blocks of memory:

- → **ROM**: This memory block is mapped to address space location 0 and has a size of 32 bytes.
- → **FLASH**: This memory block is mapped to address space location 0x40 and has a size of 4,096 bytes.
- → **RAM**: The RAM block starts at address 0x10000 and has a size of 65,536 bytes.

```
MEMORY {
    area-name : org = start-address, len = number-of-bytes
   …
}
```

```
SECTION {
      output-section-name : { contents } > area-name
      …
      GROUP {
              [ALIGN(expression)]
             section-definition
             …
      } > area-name
}
```



→ Translating this memory map into the MEMORY directive.

```
/* Memories definition -> Label_Name(AttributeList) : ORIGIN = StartMemoryAddress,   LENGTH = MemoryLength */
MEMORY
{
  CCMRAM (xrw)    : ORIGIN = 0x10000000,   LENGTH = 64K
  RAM    (xrw)    : ORIGIN = 0x20000000,   LENGTH = 128K
  FLASH  (rx)     : ORIGIN = 0x08000000,   LENGTH = 1024K
}
```

`R'
Read-only sections.
`W'
Read/write sections.
`X'
Sections containing executable code.

→ On the other hand, the SECTION directive guides the linker in determining which input sections should be combined into specific output sections. It provides instructions for the merging of sections. The general notation of the SECTION/SECTIONS command is shown.

```
SECTION {
      output-section-name : { contents } > area-name
      …
      GROUP {
              [ALIGN(expression)]
             section-definition
             …
      } > area-name
}
```

```
/* Sections */
SECTIONS{
  /* .text section, The program code and
  other data into "FLASH" Rom type memory */
  .text :
  {
    . = ALIGN(4);
    *(.text)           /* .text sections (code) */
    *(.text*)          /* .text* sections (code) */
    . = ALIGN(4);
    _etext = .;        /* Define a global symbols at end of code */
  } >FLASH

  /* .rodata section, Constant data into "FLASH" Rom type memory */
  .rodata :
  {
    . = ALIGN(4);
    *(.rodata)         /* .rodata sections (constants, strings, etc.) */
    *(.rodata*)        /* .rodata* sections (constants, strings, etc.) */
    . = ALIGN(4);
  } >FLASH

  /* .data section, Initialized data sections into "RAM" Ram type memory */
  .data :
  {
    . = ALIGN(4);
    _sdata = .;        /* Create a global symbol at data start */
    *(.data)           /* .data sections */
    *(.data*)          /* .data* sections */
    . = ALIGN(4);
    _edata = .;        /* Define a global symbol at data end */
  } >RAM AT> FLASH
```

```
  /* .bss section, Uninitialized data section into "RAM" Ram type memory */
  .bss :
  {
    _sbss = .;         /* Define a global symbol at bss start */
    *(.bss)            /* .bss sections */
    *(.bss*)           /* .bss* sections */
    . = ALIGN(4);
    _ebss = .;         /* Define a global symbol at bss end */
  } >RAM

  /* .User_heap_stack section, used to check that there is enough "RAM" Ram
  ._user_heap_stack :
  {
    . = ALIGN(8);
    . = . + _Min_Heap_Size;
    . = . + _Min_Stack_Size;
    . = ALIGN(8);
  } >RAM
}
```

```
>arm-none-eabi-objdump -t app.o
```

```
SYMBOL TABLE:
00008000 l    d  .text  00000000 .text
00018088 l    d  .data  00000000 .data
00018098 l    d  .bss   00000000 .bss
00000000 l    d  .comment       00000000 .comment
00000000 l    d  .ARM.attributes        00000000 .ARM.attributes
000180a8 l    d  .noinit        00000000 .noinit
00000000 l    df *ABS*  00000000 file1.c
00000000 l    df *ABS*  00000000 file2.c
00018090 g     O .data  00000004 var5
000180a8 g       .bss   00000000 _bss_end__
00018098 g       .bss   00000000 __bss_start__
00018088 g     O .data  00000004 var1
0001809c g     O .bss   00000004 var4
000180a4 g     O .bss   00000004 var8
000180a8 g       .bss   00000000 __bss_end__
00000000         *UND*  00000000 _start
00008044 g     F .text  00000044 fun2
00018098 g       .bss   00000000 __bss_start
0001808c g     O .data  00000004 var2
000180a8 g       .bss   00000000 __end__
00018094 g     O .data  00000004 var6
00018098 g     O .bss   00000004 var3
000180a0 g     O .bss   00000004 var7
00018098 g       .data  00000000 _edata
000180a8 g       .bss   00000000 _end
00008000 g     F .text  00000044 fun1
00080000 g       .comment       00000000 _stack
00018088 g       .data  00000000 __data_start
```

```
>arm-none-eabi-ld file1.o file2.o -o application.o -T LinkerScript.ld
```

```
SYMBOL TABLE:
08000000 l    d  .text  00000000 .text
20000000 l    d  .data  00000000 .data
20000010 l    d  .bss   00000000 .bss
00000000 l    d  .comment       00000000 .comment
00000000 l    d  .ARM.attributes        00000000 .ARM.attributes
00000000 l    df *ABS*  00000000 file1.c
00000000 l    df *ABS*  00000000 file2.c
20000008 g     O .data  00000004 var5
20000000 g     O .data  00000004 var1
20000014 g     O .bss   00000004 var4
2000001c g     O .bss   00000004 var8
08000044 g     F .text  00000044 fun2
20000004 g     O .data  00000004 var2
2000000c g     O .data  00000004 var6
20000010 g     O .bss   00000004 var3
20000018 g     O .bss   00000004 var7
08000000 g     F .text  00000044 fun1
```

→ **Section alignment with the linker:**

- During the linking process, the GCC linker ensures that input sections are properly aligned by inserting padding, if necessary. This padding guarantees that each section starts at an address that is a multiple of its alignment requirement. By aligning the sections correctly, the linker optimizes memory access and ensures the proper execution of the program.

```
/* .data section, Initialized data sections into "RAM" Ram type memory */
.data :
{
  . = ALIGN(4);
  *(.data)          /* .data sections */
  *(.data*)         /* .data* sections */
  . = ALIGN(4);
} >RAM
```

```
unsigned int var1;
unsigned short var2;
unsigned char var3;
unsigned int var4;
```

var1	var1	var1	var1
var2	var2	-	-
var3	-	-	-
var4	var4	var4	var4

→ **user define section ➔ this is done by using GCC attribute**

→ **__attribute__ ((section ("Section_Name")))**

`section ("section-name")`

Normally, the compiler places the code it generates in the `text` section. Sometimes, however, you need additional sections, or you need certain particular functions to appear in special sections. The `section` attribute specifies that a function lives in a particular section. For example, the declaration:

```
extern void foobar (void) __attribute__ ((section ("bar")));
```

puts the function `foobar` in the `bar` section.

Some file formats do not support arbitrary sections so the `section` attribute is not available on all platforms. If you need to map the entire contents of a module to a particular section, consider using the facilities of the linker instead.

`section ("section-name")`

Normally, the compiler places the objects it generates in sections like `data` and `bss`. Sometimes, however, you need additional sections, or you need certain particular variables to appear in special sections, for example to map to special hardware. The `section` attribute specifies that a variable (or function) lives in a particular section. For example, this small program uses several specific section names:

```
struct duart a __attribute__ ((section ("DUART_A"))) = { 0 };
struct duart b __attribute__ ((section ("DUART_B"))) = { 0 };
char stack[10000] __attribute__ ((section ("STACK"))) = { 0 };
int init_data __attribute__ ((section ("INITDATA")));
```

```
__attribute__ ((section (".ali")))
unsigned int var4;

__attribute__ ((section (".ahmed")))
void fun1(void){
    var1++;
    var2++;
    var3++;
    var4++;
    fun2();
}
```

Function attribute

arm-none-eabi-objdump -t file1.o

```
file1.o:     file format elf32-littlearm

SYMBOL TABLE:
00000000 l    df *ABS*  00000000 file1.c
00000000 l    d  .text  00000000 .text
00000000 l    d  .data  00000000 .data
00000000 l    d  .bss   00000000 .bss
00000000 l    d  .ali   00000000 .ali
00000000 l    d  .ahmed 00000000 .ahmed
00000000 l    d  .comment       00000000 .comment
00000000 l    d  .ARM.attributes        00000000 .ARM.attributes
00000000 g     O .data  00000004 var1
00000004 g     O .data  00000002 var2
00000000 g     O .bss   00000001 var3
00000000 g     O .ali   00000004 var4
00000000 g     F .ahmed 00000048 fun1
00000000         *UND*  00000000 fun2
```

variable attribute

```
/* Sections */
SECTIONS{
  /* .text section, The program code and other data into "FLASH" Rom type memory */
  .text :
  {
    . = ALIGN(4);
    *(.text)           /* .text sections (code) */
    *(.ahmed)
    *(.text*)          /* .text* sections (code) */
    . = ALIGN(4);
  } >FLASH
```

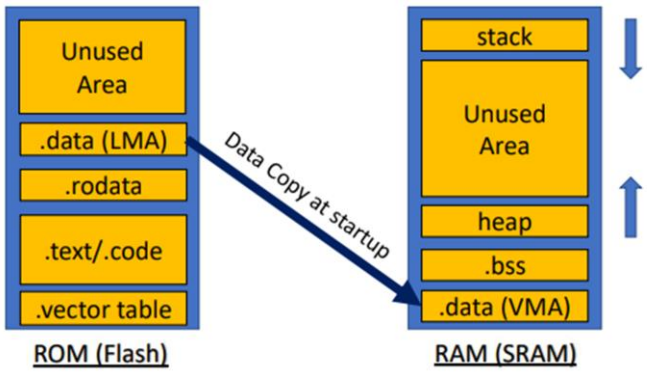
→ The **.data** section will be copied by **the startup** code from FLASH to SRAM.

→ This type of section has:

- **LMA** → Load Memory Address
- **VMA** → Virtual Memory Address

→ It is necessary to specify in the linker script which section will be copied from Load Memory Address (LMA) to Virtual Memory Address (VMA).

```
/* Sections */
SECTIONS{
  .section :
  {

  } >VMA AT> LMA
}

/* .data section, Initialized data sections into "RAM" Ram type memory */
.data :
{
  . = ALIGN(4);
  *(.data)           /* .data sections */
  *(.data*)          /* .data* sections */
  . = ALIGN(4);
} >RAM AT> FLASH
```

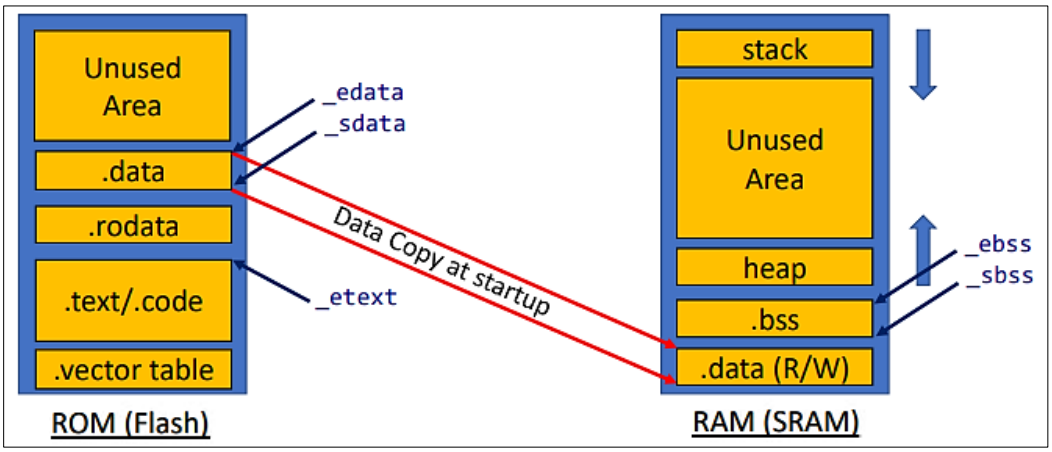


→ Linker will generate load address at FLASH and absolute memory address at RAM.

→ The startup code needs a help from the linker script file to define the boundaries of the .data section (The start address of .data section (end address of .rodata) and the length of the .data section.)

→ To do that we will introduce the concept of (**Location Counter**) and (**Linker Symbols**).

→ The startup code will use these symbols to do the data copy and BSS initialization.



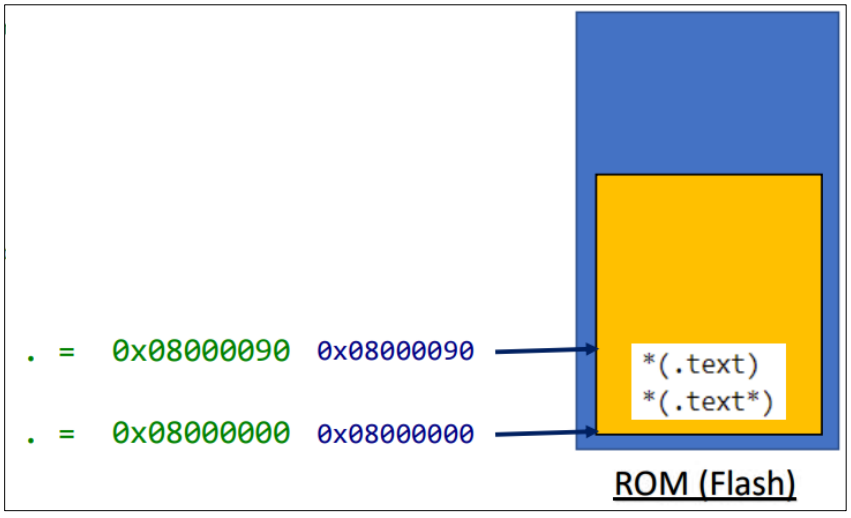
→ There is a difference between the concept of variable declaration and symbol declaration.

→ The symbol is just a name of an address.

→ **the location counter:**

- The special linker variable dot (.) always contains the current output location counter.
- Since the (.) always refers to a location in an output section, it may only appear in an expression within a SECTIONS command.
- Assigning a value to (.) will cause the location counter to be moved.
  - This may be used to create holes in the output section.
- The location counter may never be moved backwards.

```
/* .data section, Initialized data sections into "RAM" Ram type memory */
.data :
{
  . = ALIGN(4);
  _sdata = .;        /* Create a global symbol at data start */
  *(.data)           /* .data sections */
  *(.data*)          /* .data* sections */
  . = ALIGN(4);
  _edata = .;        /* Define a global symbol at data end */
} >RAM AT> FLASH

/* .bss section, Uninitialized data section into "RAM" Ram type memory */
.bss :
{
  _sbss = .;         /* Define a global symbol at bss start */
  *(.bss)            /* .bss sections */
  *(.bss*)           /* .bss* sections */
  . = ALIGN(4);
  _ebss = .;         /* Define a global symbol at bss end */
} >RAM
```




```
/* Highest address of the user mode stack */
_estack = ORIGIN(RAM) + LENGTH(RAM); /* End of "RAM" Ram type memory */

_Min_Heap_Size = 0x200 ; /* Required amount of heap */
_Min_Stack_Size = 0x400 ; /* Required amount of stack */
```

→ **_estack**: This symbol defines the highest address of the stack. It is calculated by adding the origin address of the RAM region to its length. the stack typically grows downwards from higher memory addresses to lower memory addresses. Therefore, _estack represents the topmost address of the stack in RAM.

→ **Min_Heap_Size**: This symbol specifies the required amount of heap memory for dynamic memory allocation. In this case, **_Min_Heap_Size** is set to **0x200** bytes, indicating that at least **512** bytes of heap memory are required.

→ **Min_Stack_Size**: This symbol indicates the required amount of stack memory for program execution. In this case, **_Min_Stack_Size** is set to **0x400** bytes, indicating that at least **1024** bytes of stack memory are required for the program.

→ By defining the stack and heap sizes, developers can ensure that sufficient memory is allocated for program execution and dynamic memory allocation, thus preventing stack overflows and heap fragmentation issues.

```
/* .User_heap_stack section, used to check that there is enough "RAM" Ram  type memory left */
._user_heap_stack :
{
  . = ALIGN(8);
  . = . + _Min_Heap_Size;
  . = . + _Min_Stack_Size;
  . = ALIGN(8);
} >RAM
```

➢ **the MAP file:**

→ Firmware engineers rarely reach for the map file generated by their build process when debugging.

→ The map file provides valuable information that can help you understand and optimize memory.

→ The map file is a symbol table for the whole program.

→ As you see all linker script symbols, variables and functions are in the .map file with their absolute addresses.

```
Memory Configuration

Name             Origin             Length             Attributes
CCMRAM           0x10000000         0x00010000         xrw
RAM              0x20000000         0x00020000         xrw
FLASH            0x08000000         0x00100000         xr
*default*        0x00000000         0xffffffff

Linker script and memory map

                0x20020000                _estack = (ORIGIN (RAM) + LENGTH (RAM))
                0x00000200                _Min_Heap_Size = 0x200
                0x00000400                _Min_Stack_Size = 0x400

.data           0x20000000        0x10 load address 0x08000090
                0x20000000                . = ALIGN (0x4)
                0x20000000                _sdata = .
*(.data)
.data           0x20000000         0x6 file1.o
                0x20000000                var1
                0x20000004                var2
*fill*          0x20000006         0x2
.data           0x20000008         0x8 file2.o
                0x20000008                var5
                0x2000000c                var6
*(.data*)
                0x20000010                . = ALIGN (0x4)
                0x20000010                _edata = .
```

→ **How to calculate the size of .data section?**

```
.data           0x20000000        0x10 load address 0x08000090
                0x20000000                . = ALIGN (0x4)
                0x20000000                _sdata = .
*(.data)
.data           0x20000000         0x6 file1.o
                0x20000000                var1
                0x20000004                var2
*fill*          0x20000006         0x2
.data           0x20000008         0x8 file2.o
                0x20000008                var5
                0x2000000c                var6
*(.data*)
                0x20000010                . = ALIGN (0x4)
                0x20000010                _edata = .
```

```
unsigned char var3 = 0;
unsigned int var4;
unsigned int var7 = 0;
unsigned int var8;
```

- .data section size = _edata - _sdata
- Size = 0x20000010 – 0x20000000 = 0x10 = 16 Bytes

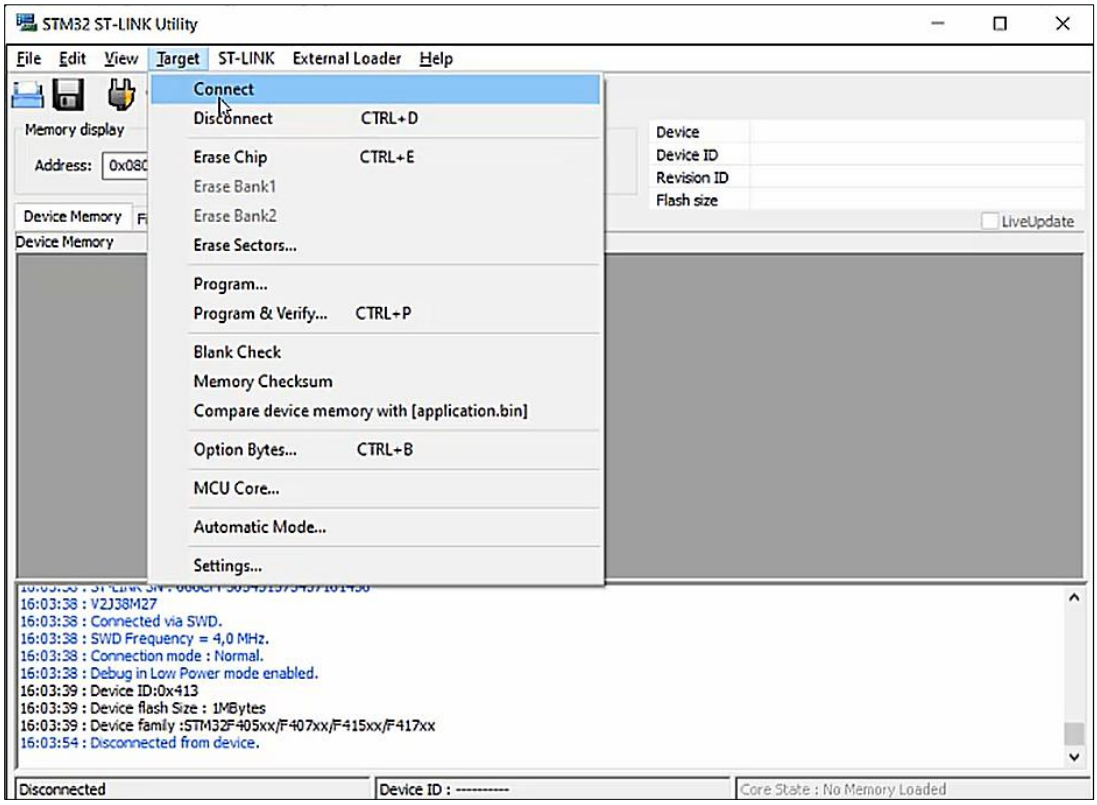
→ **To generate the MAP file ➔ using our liker script file**

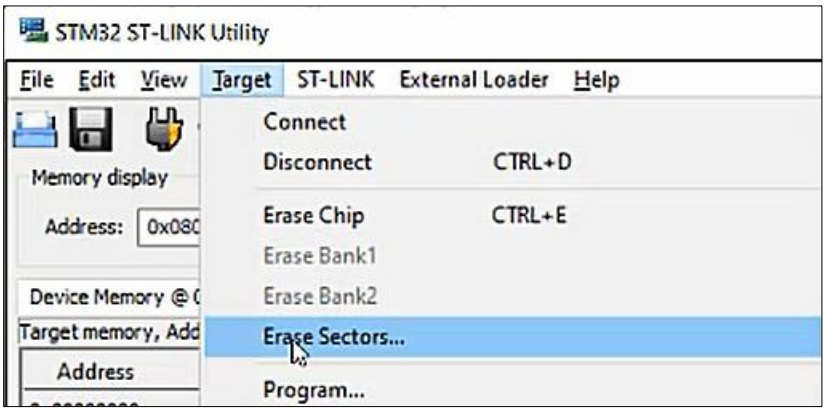
```
>arm-none-eabi-ld file1.o file2.o -o application.o -Map application.map -T LinkerScript.ld
```

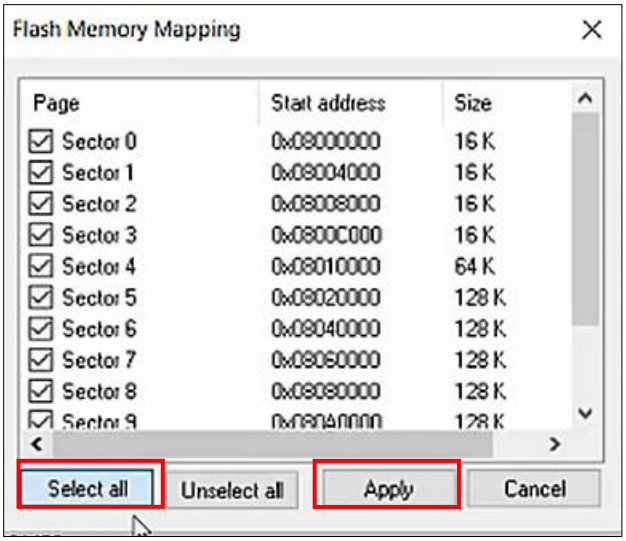
→ **To generate the (.elf / .bin / .hex) file**

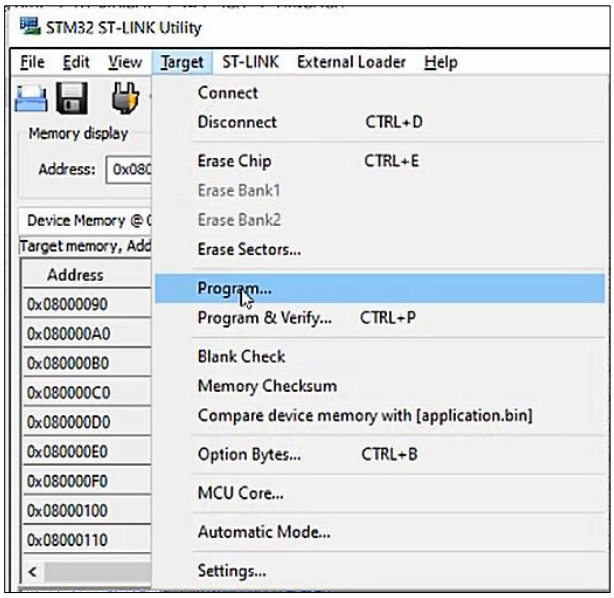
```
>arm-none-eabi-ld -Map application.map -T LinkerScript.ld file1.o file2.o -o application.elf

>arm-none-eabi-objcopy -O binary application.elf application.bin

>arm-none-eabi-objcopy -O ihex application.elf application.hex
```

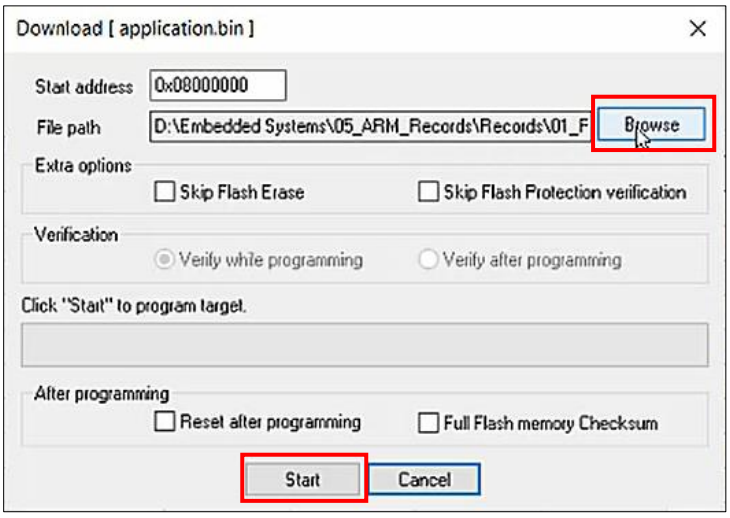
→ **To flash on of these file on the discovery board**



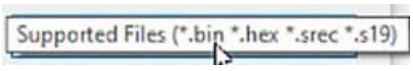


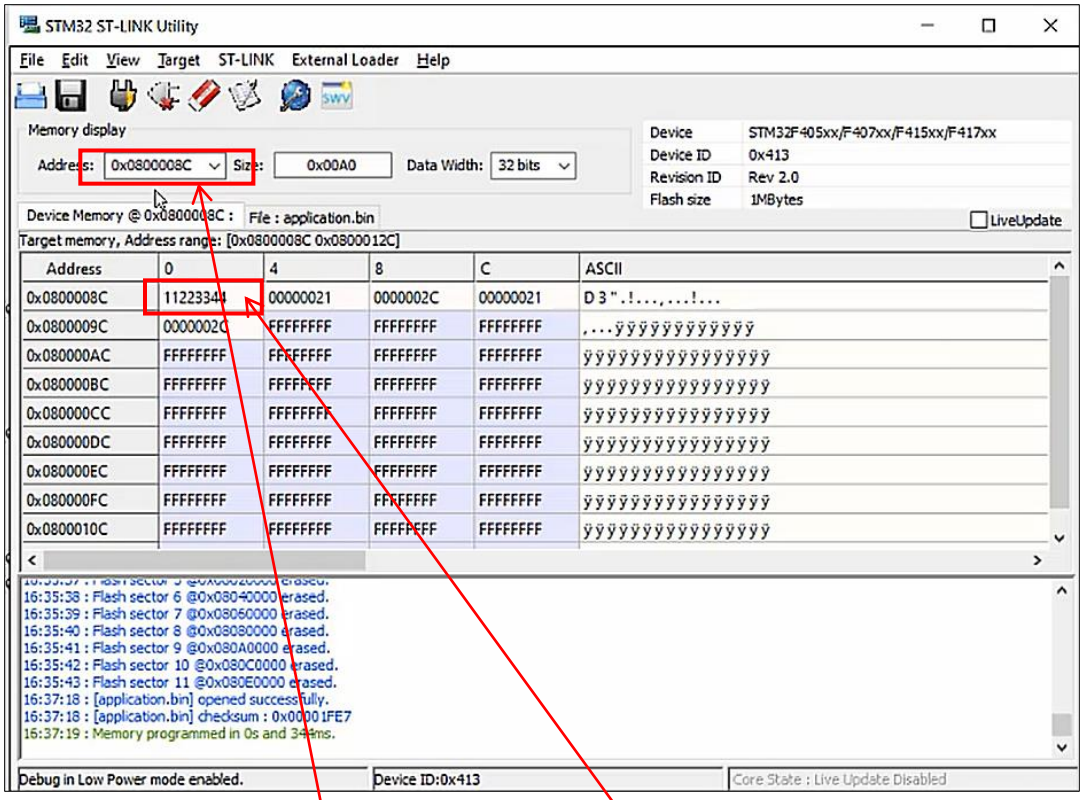






- **Then we choose the (.bin / .hex) file to flash on the board then press start**






```
const unsigned int var11 = 0x11223344;
```

```
.rodata         0x0800008c      0x4
                0x0800008c              . = ALIGN (0x4)
 *(.rodata)
 .rodata        0x0800008c      0x4 file1.o
                0x0800008c              var11
 *(.rodata*)
                0x08000090              . = ALIGN (0x4)
```

THE END