How File Execution Happens in Embedded Systems?

In embedded systems, program execution begins only after a series of file transformations performed during cross compilation. The source code written in a high-level language is progressively converted into lower-level representations until a target-executable file is produced.

The build process starts with preprocessing, where headers and macros are expanded. The compiler then translates the processed code into assembly instructions specific to the target architecture. These instructions are assembled into object files containing machine code. Multiple object files, along with the startup file, are finally linked using a linker script to form an executable file (.elf). When the system powers on or resets, execution begins from the reset vector defined in the startup code, which initializes memory and transfers control to the main() function.

Role of Cross Compiler in File Execution:

In this system, the development machine (host) and the embedded controller (target) use different CPU architectures.

- The host system is based on x86 / x86-64 architecture (desktop or laptop PC).
- The target system is based on ARM architecture (microcontroller).

Since binaries compiled for x86 cannot execute on an ARM processor, a cross compiler is required.

A cross compiler runs entirely on the x86 host system, but generates machine code that is compatible with the ARM target architecture. All compilation stages—preprocessing, compilation, assembling, and linking—are executed on the host, while the final executable is intended to run only on the target controller.

The generated output file (such as .elf) contains ARM instructions, startup routines, and memory mappings defined specifically for the target hardware. This ensures correct execution once the program is flashed onto the controller.

What commands does the Compile button actually run?

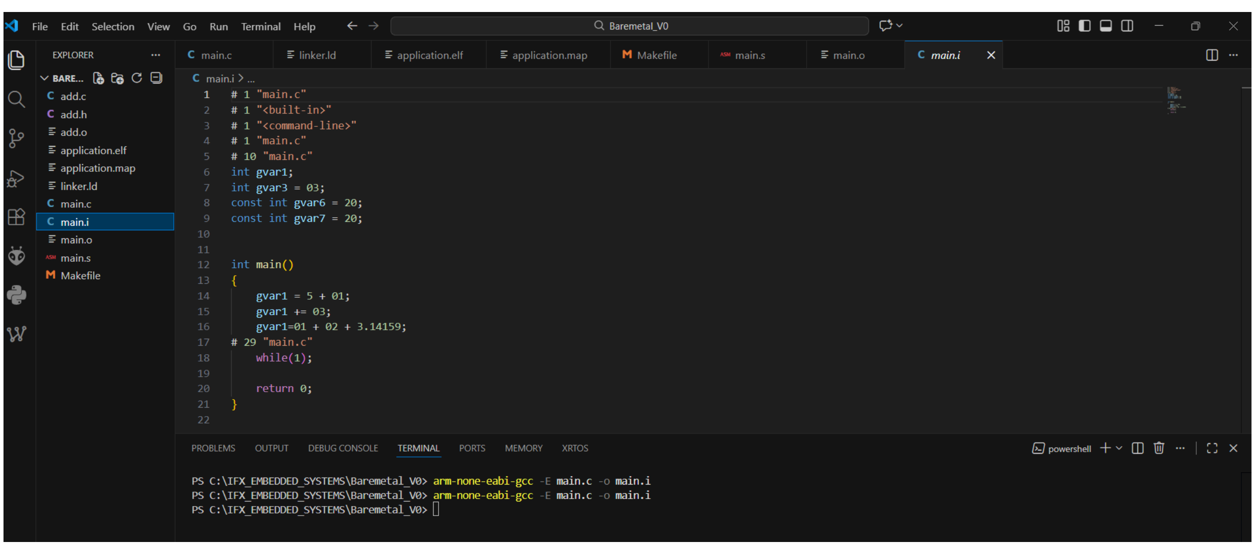
After the compilation button is pressed the following process takes place:

STEP 1: Preprocessing (.c → .i)

```
arm-none-eabi-gcc -E main.c -o main.i
```

What happens

- Header files are expanded
- Macros are replaced
- Comments are removed



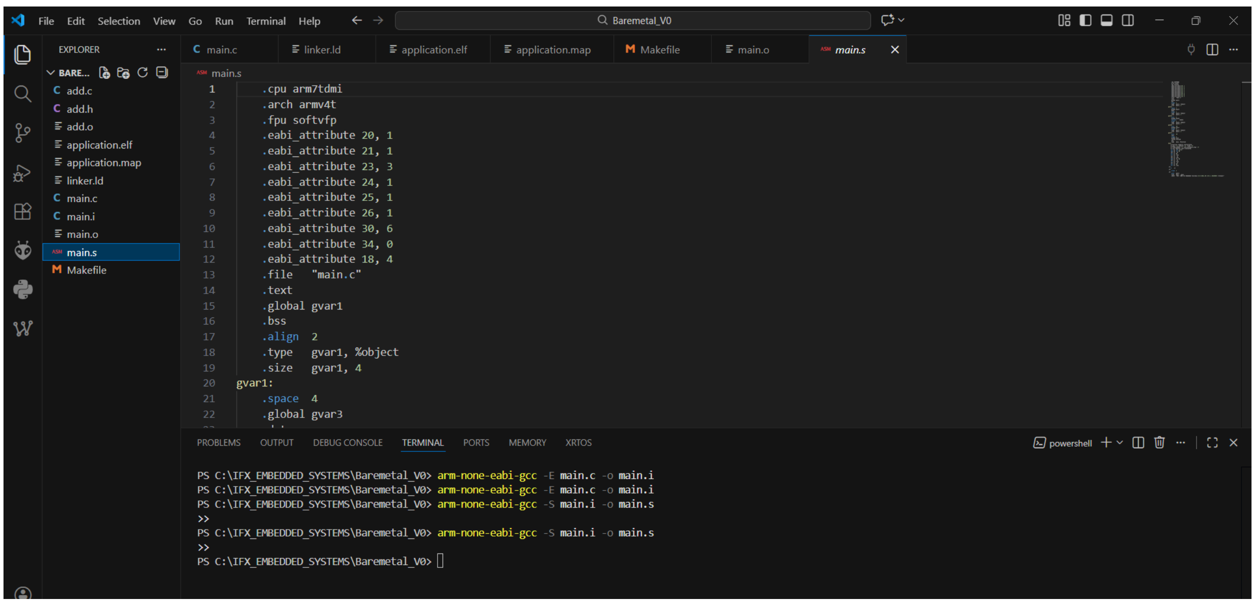
OUTPUT FILE: main.i

STEP 2: Compilation (.i → .s)

```
arm-none-eabi-gcc -S main.i -o main.s
```

What happens

- C code is converted into ARM assembly
- Optimizations are applied
- Register usage is decided



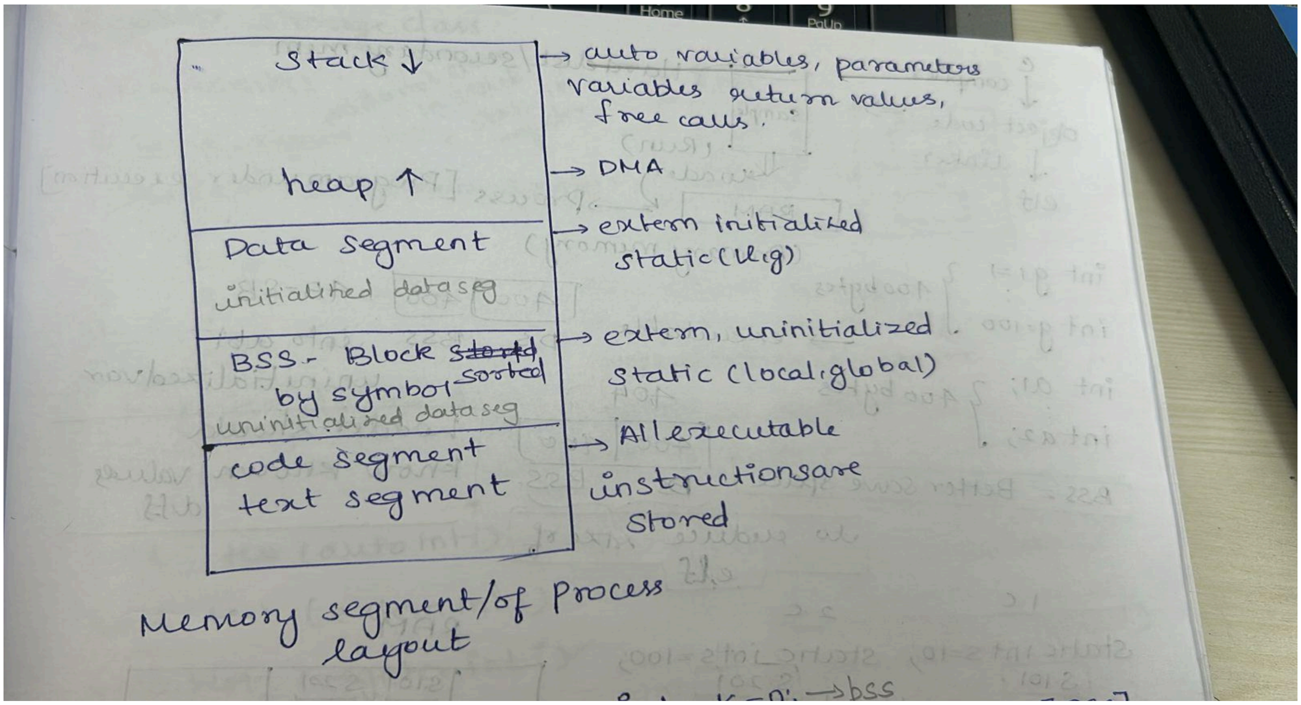
OUTPUT FILE: main.s

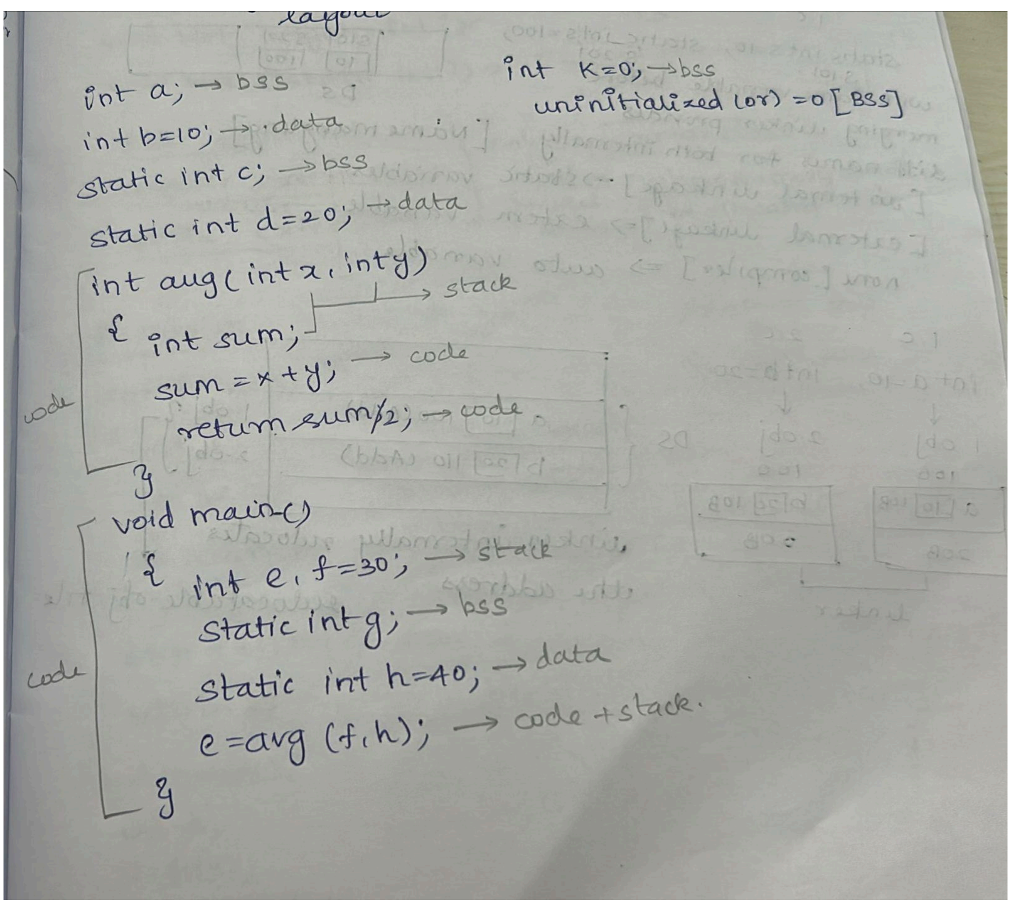
STEP 3: Assembling (.s → .o)

```
arm-none-eabi-gcc -c main.s -o main.o
```

What happens

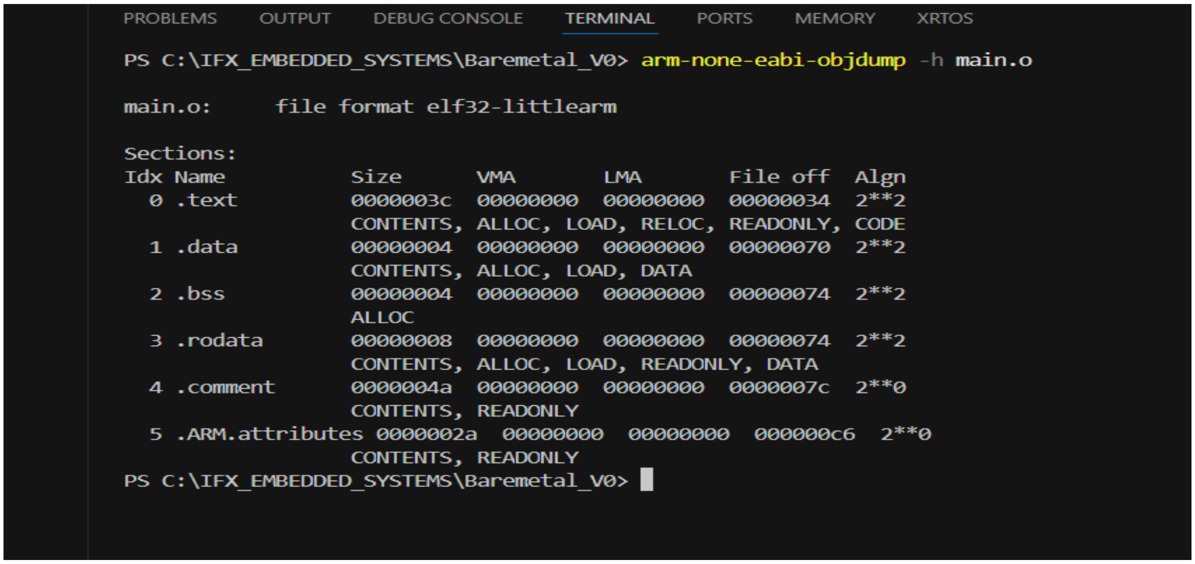
- Assembly is converted to ARM machine code
- Sections like .text, .data, .bss are created
- Symbols remain unresolved





To visualize how data is stored in memory segments:

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



```
PROBLEMS   OUTPUT   DEBUG CONSOLE   TERMINAL   PORTS   MEMORY   XRTOS

PS C:\IFX_EMBEDDED_SYSTEMS\Baremetal_V0> arm-none-eabi-objdump -h main.o

main.o:     file format elf32-littlearm

Sections:
Idx Name          Size      VMA       LMA       File off  Algn
  0 .text         0000003c  00000000  00000000  00000034  2**2
                  CONTENTS, ALLOC, LOAD, RELOC, READONLY, CODE
  1 .data         00000004  00000000  00000000  00000070  2**2
                  CONTENTS, ALLOC, LOAD, DATA
  2 .bss          00000004  00000000  00000000  00000074  2**2
                  ALLOC
  3 .rodata       00000008  00000000  00000000  00000074  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  4 .comment      0000004a  00000000  00000000  0000007c  2**0
                  CONTENTS, READONLY
  5 .ARM.attributes 0000002a  00000000  00000000  000000c6  2**0
                  CONTENTS, READONLY
PS C:\IFX_EMBEDDED_SYSTEMS\Baremetal_V0>
```

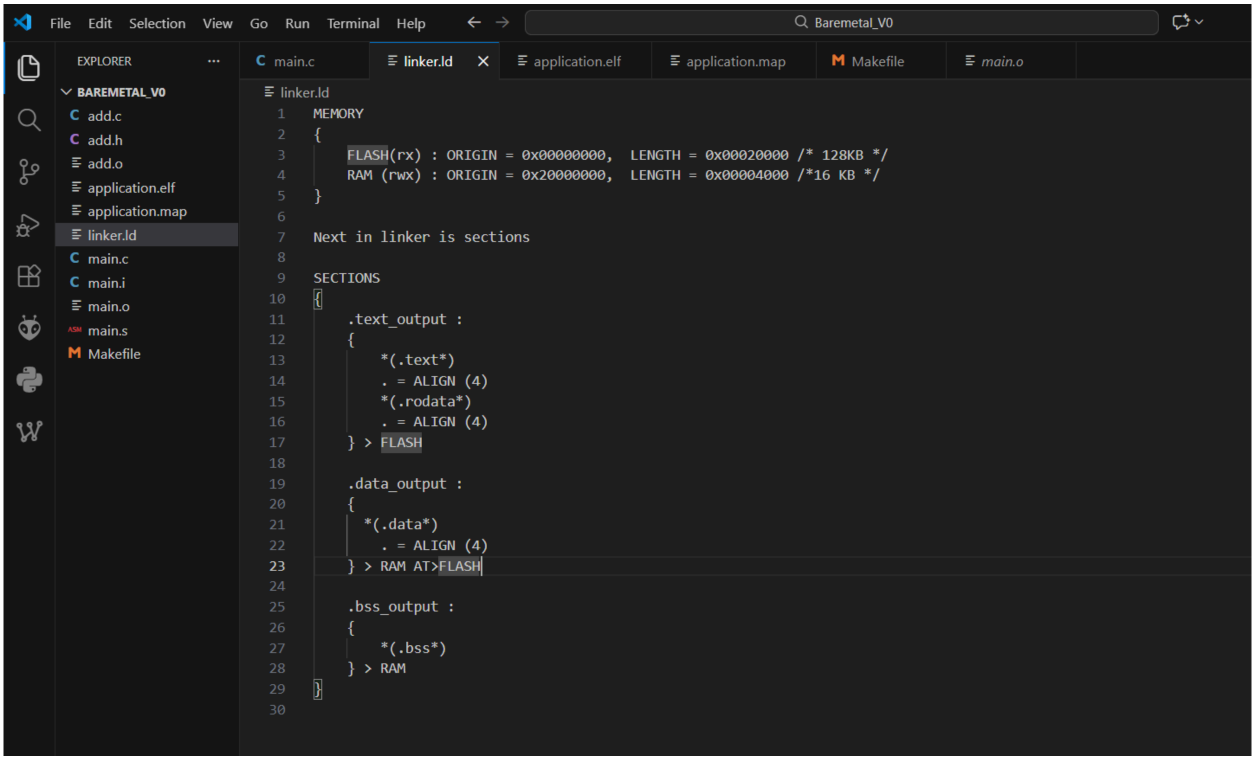
What Is a Linker Script?

A linker script is a text file that tells the linker how to:

- Arrange program sections in memory
- Assign absolute addresses
- Map logical sections to physical memory (Flash, RAM)

In bare-metal embedded systems, the linker script controls where every byte of code and data resides.

Without a linker script, the program cannot execute correctly on the target hardware.



VMA vs LMA in Linker Script

What Are VMA and LMA?

In embedded systems, some program sections are stored in one memory region but executed from another.
To handle this, the linker uses two different addresses:

- VMA (Virtual Memory Address)
  → Address where the section resides during execution

- LMA (Load Memory Address)
  → Address where the section is stored in non-volatile memory

SECTION	LMA(STORED IN)	VMA(EXECUTED IN)
.text	Flash	Flash
.rodata	Flash	Flash
.data	Flash	RAM
.bss	Cleared in RAM	RAM

```
SECTIONS
{
    .text_output :
    {
        *(.text*)
        . = ALIGN (4)
        *(.rodata*)
        . = ALIGN (4)
    } > FLASH

    .data_output :
    {
      *(.data*)
        . = ALIGN (4)
    } > RAM AT>FLASH

    .bss_output :
    {
        *(.bss*)
    } > RAM
}
```

`> RAM` → **VMA** (runtime location)

`AT > FLASH` → **LMA** (load location)

Linker is used to link all the object files present.

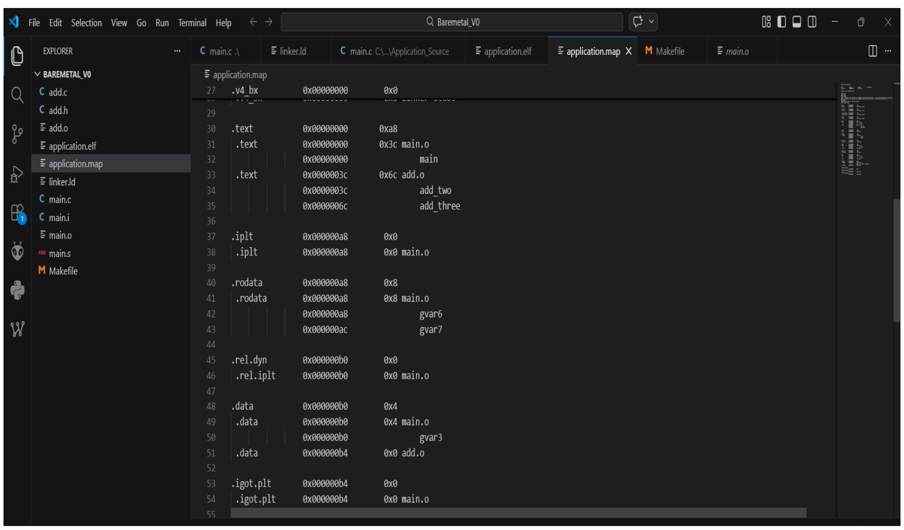
```
arm-none-eabi-gcc -c main.c -o main.o
arm-none-eabi-gcc -c add.c -o add.o
```

Next we are creating executable file by combining , all .o files

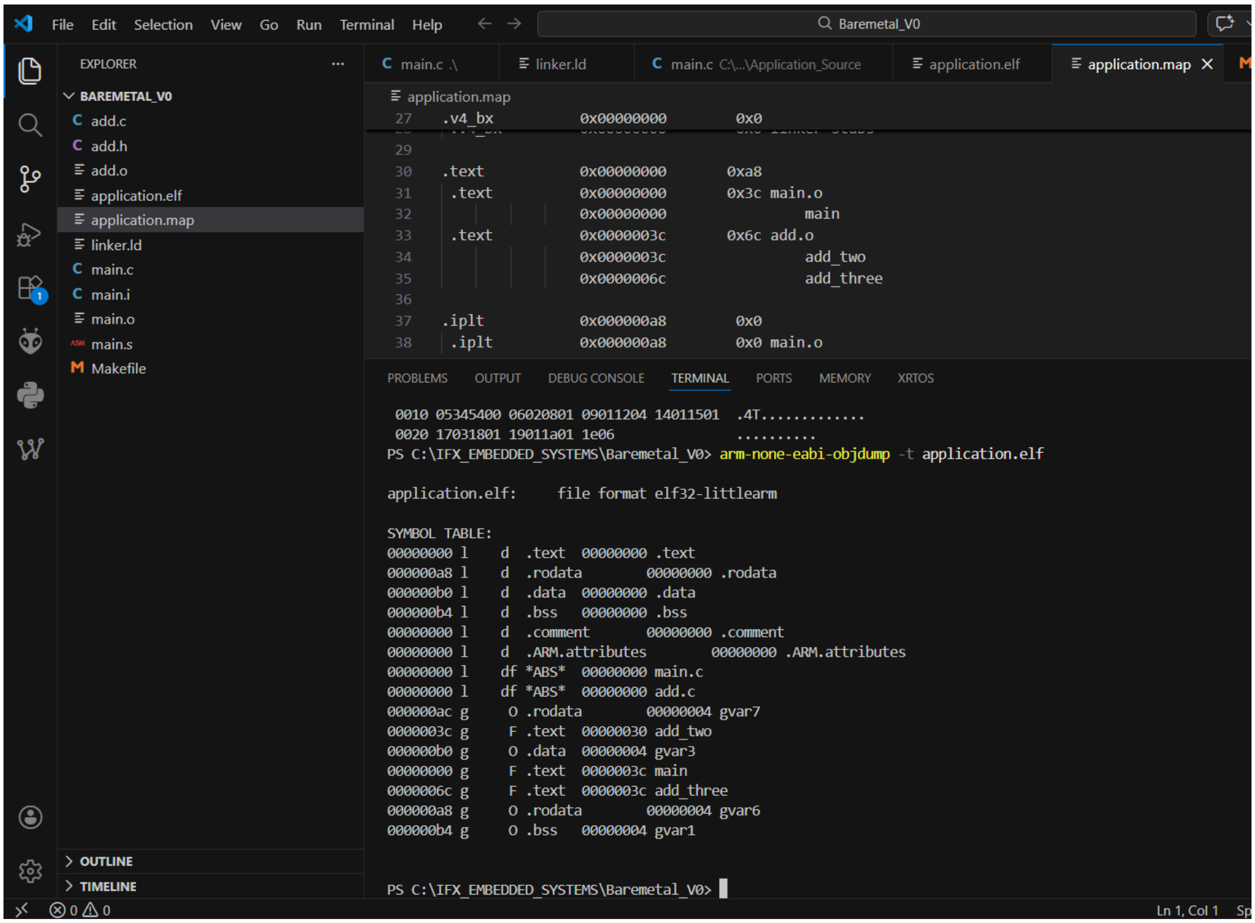
```
arm-none-eabi-gcc -T linker.ld main.o add.o -o application.elf -nostartfiles
```

-nostartfiles states it is a sample project no start files are included

A new file called application.elf will be generated
Where Application.map file shows how data are stored in each section in how much memory



```
arm-none-eabi-objdump -t application.elf
```



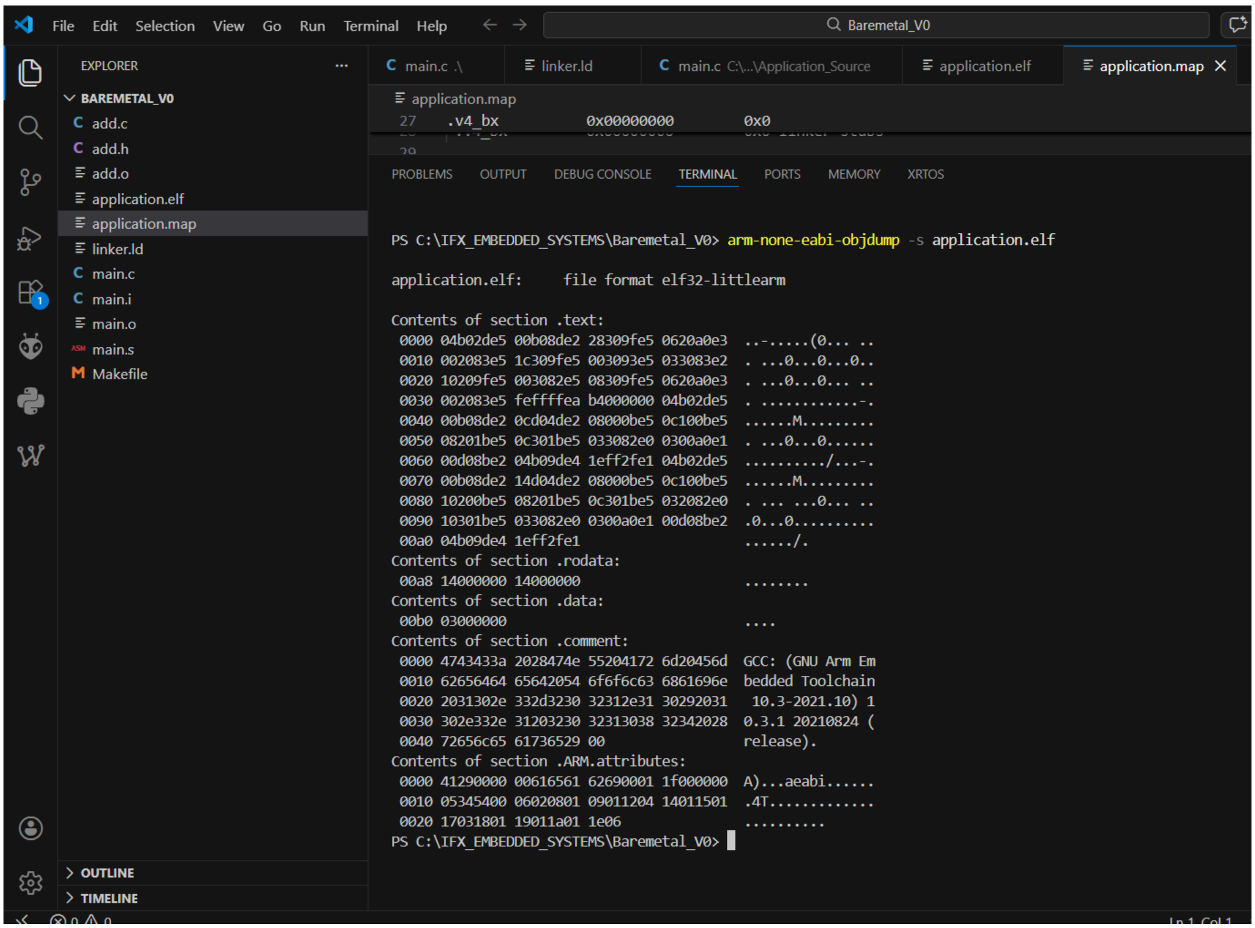

```
arm-none-eabi-objdump -t application.elf
```


This command displays the symbol table of the executable file application.elf.

The symbol table contains information about:

- Functions
- Global variables

- Static variables
- Linker-generated symbols
- Their addresses and sizes in memory




```
arm-none-eabi-objdump -s application.elf
```

The arm-none-eabi-objdump -s command displays the raw byte-level contents of each section in an ELF file, allowing developers to inspect the actual machine code and initialized data that will reside in memory.

What is Makefile?

A Makefile is a build automation file used by the make tool to compile, assemble, and link source files automatically.

In embedded systems, a Makefile replaces the IDE Compile / Build button by explicitly defining:

- Which files to compile
- Which compiler to use
- How object files are linked
- How the final executable is generated