

## 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?

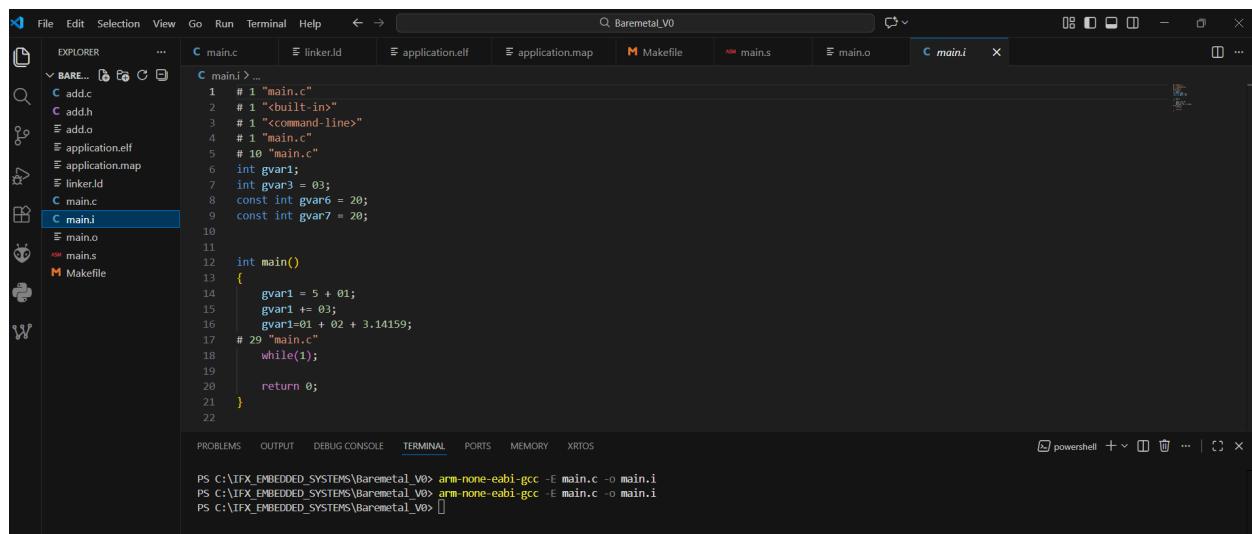
After the compilation button is pressed the following process takes place:

### 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



```
PS C:\TFX_EMBEDDED_SYSTEMS\Baremetal_V0> arm-none-eabi-gcc -E main.c -o main.i
PS C:\TFX_EMBEDDED_SYSTEMS\Baremetal_V0> arm-none-eabi-gcc -E main.c -o main.i
PS C:\TFX_EMBEDDED_SYSTEMS\Baremetal_V0> [ ]
```

The screenshot shows a terminal window with the command `arm-none-eabi-gcc -E main.c -o main.i` being run twice. The output is shown in the terminal window below the command line.

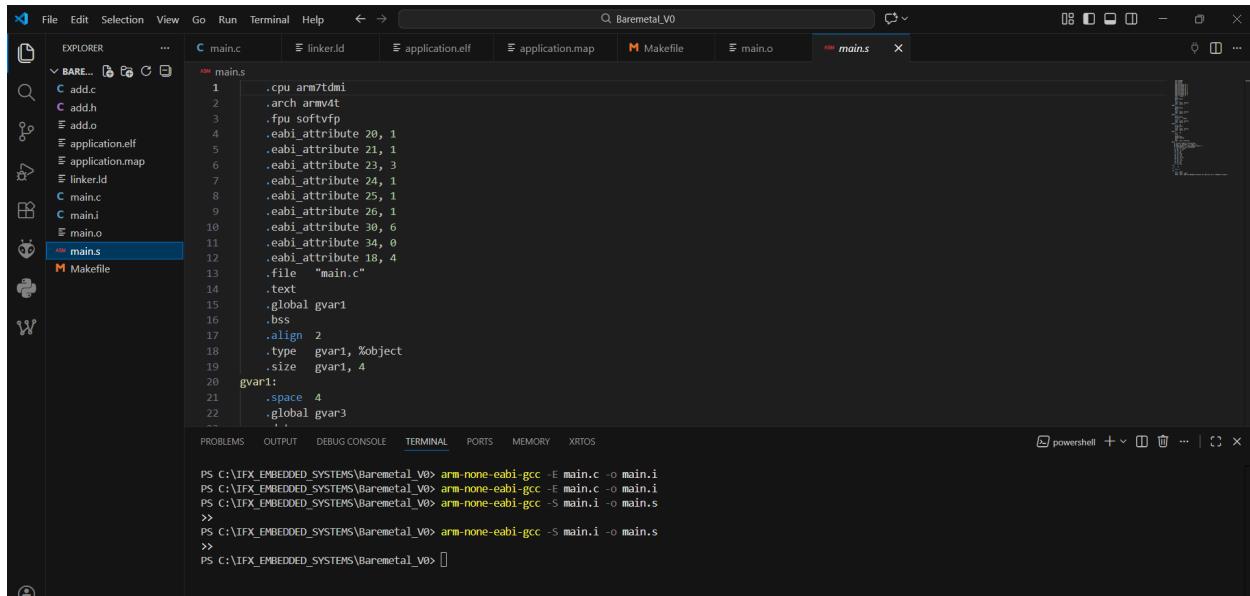
### OUTPUT FILE: main.i

### 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



The screenshot shows a terminal window with the following content:

```
Baremetal_V0
EXPLORER File Edit Selection View Go Run Terminal Help ← → Baremetal_V0
main.c linker.ld application.elf application.map Makefile main.o main.s
add.c add.h add.o application.elf application.map linker.ld main.c main.i main.o mains Makefile
CPU arm7tdmi
.arch armv4t
.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, 0
.eabi_attribute 18, 4
.file "main.c"
.text
.global gvar1
.bss
.align 2
.type gvar1, %object
.size gvar1, 4
gvar1:
.space 4
.global gvar3
.

PS C:\IFX_EMBEDDED_SYSTEMS\Baremetal_V0> arm-none-eabi-gcc -E main.c -o main.i
PS C:\IFX_EMBEDDED_SYSTEMS\Baremetal_V0> arm-none-eabi-gcc -E main.c -o main.i
PS C:\IFX_EMBEDDED_SYSTEMS\Baremetal_V0> arm-none-eabi-gcc -S main.i -o main.s
>>
PS C:\IFX_EMBEDDED_SYSTEMS\Baremetal_V0> arm-none-eabi-gcc -S main.i -o main.s
>>
PS C:\IFX_EMBEDDED_SYSTEMS\Baremetal_V0>
```

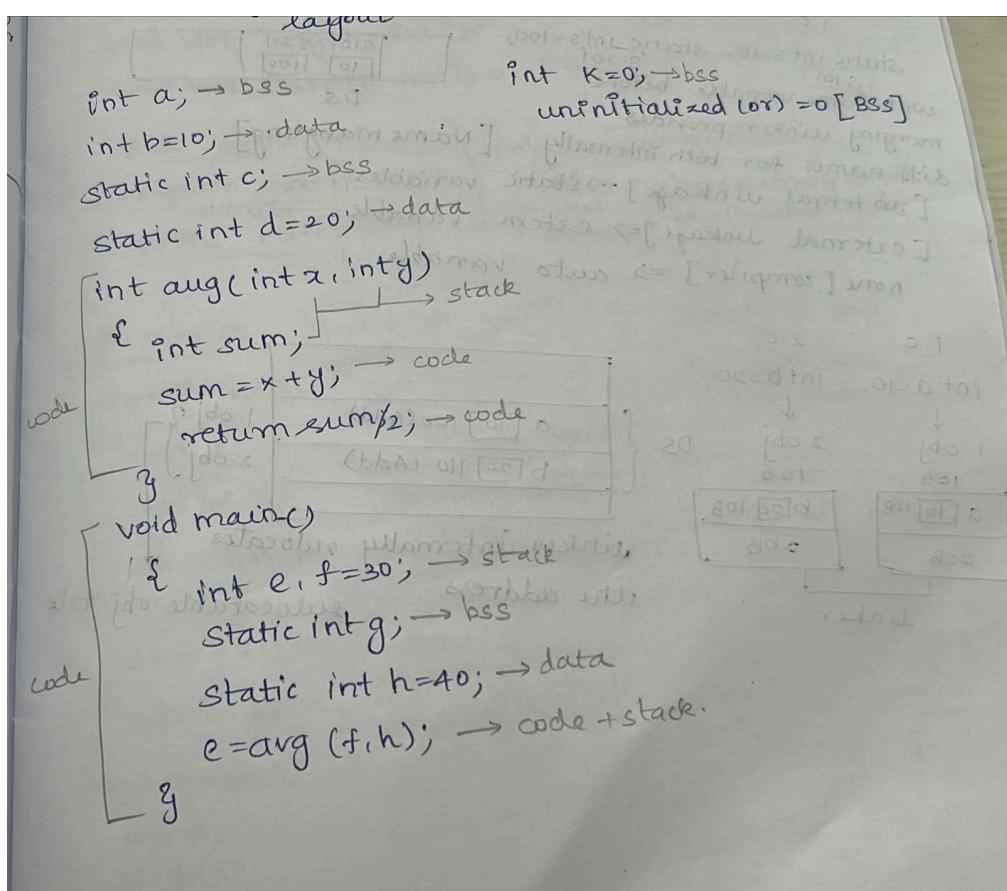
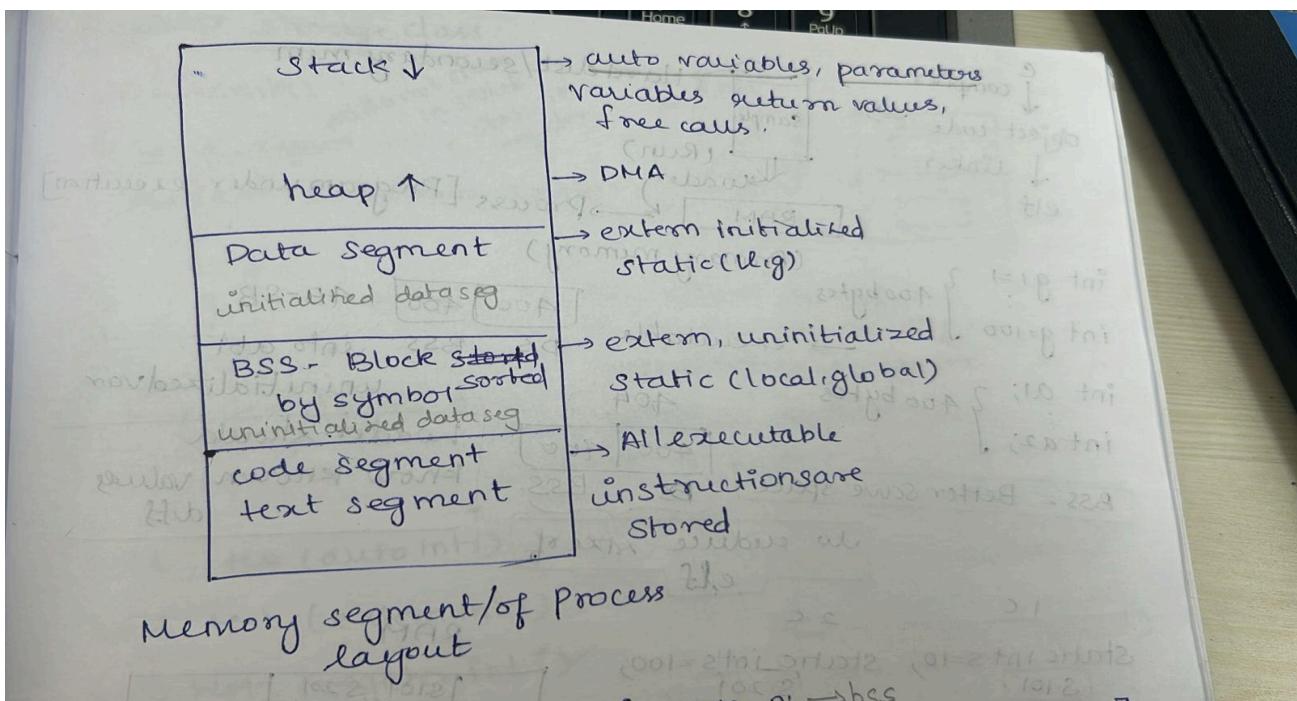
## OUTPUT FILE: main.s

**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
```

```
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  Align
 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.**

The screenshot shows a terminal window with the title "Baremetal\_V0". The window has tabs for "main.c", "linker.ld", "application.elf", "application.map", "Makefile", and "main.o". The "linker.ld" tab is active, displaying the following linker script:

```
MEMORY
{
    FLASH(rx) : ORIGIN = 0x00000000, LENGTH = 0x00020000 /* 128KB */
    RAM(rwx) : ORIGIN = 0x20000000, LENGTH = 0x00004000 /* 16 KB */
}

Next in linker is sections

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

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

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

## 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

The screenshot shows a terminal window with the following command and its output:

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

Below the terminal, the application.map file is displayed in a code editor. The map file contains memory layout information for the application.elf file, showing sections like .text, .data, and .bss, along with their addresses and sizes.

Section	Address	Size	File
.text	0x00000000	0xa8	main.o
.text	0x00000000	0xc	main.o
.text	0x00000000	main	
.text	0x0000003c	0x6c	add.o
.text	0x00000093c	add_two	
.text	0x00000096c	add_three	
.iplt	0x000000a8	0x0	
.iplt	0x000000a8	0x0	main.o
.rodata	0x000000a8	0x8	
.rodata	0x000000a8	0x8	main.o
	0x000000a8	gvar5	
	0x000000ac	gvar7	
.rel.dyn	0x000000b0	0x0	
.rel.iplt	0x000000b0	0x0	main.o
.data	0x000000b0	0x4	
.data	0x000000b0	0x4	main.o
	0x000000b0	gvar3	
.data	0x000000b4	0x0	add.o
.igot.plt	0x000000b4	0x0	
.igot.plt	0x000000b4	0x0	main.o

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

The screenshot shows the Baremetal\_V0 IDE interface. The terminal window displays the output of the command `arm-none-eabi-objdump -t application.elf`. The output includes the symbol table for the executable file, listing various symbols such as functions (.text), data (.data), and global variables (.rodata) along with their addresses and sizes.

```
application.map
27 .v4_bx      0x00000000  0x0
29
30 .text        0x00000000  0xa8
31 .text        0x00000000  0x3c main.o
32 .text        0x00000000  0x6c add.o
33 .text        0x0000003c  0x6c add_two
34 .text        0x0000003c  0x6c add_three
35 .text        0x0000006c  0x6c add_three
36
37 .iplt       0x000000a8  0x0
38 .iplt       0x000000a8  0x0 main.o

application.elf:   file format elf32-littlearm

SYMBOL TABLE:
00000000 l  d  .text  00000000 .text
00000008 l  d  .rodata  00000000 .rodata
000000b0 l  d  .data  00000000 .data
000000b4 l  d  .bss  00000000 .bss
00000000 l  d  .comment  00000000 .comment
00000000 l  d  .ARM.attributes  00000000 .ARM.attributes
00000000 l  df *ABS* 00000000 main.c
00000000 l  df *ABS* 00000000 add.c
000000ac g  o  .rodata  00000004 gvar7
0000003c g  F  .text  00000030 add_two
000000b0 g  o  .data  00000004 gvar3
00000000 g  F  .text  00000003 main
0000000c g  F  .text  0000003c add_three
000000a8 g  o  .rodata  00000004 gvar6
000000b4 g  o  .bss  00000004 gvar1
```

```
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

```

PS C:\IFX_EMBEDDED_SYSTEMS\Baremetal_V0> arm-none-eabi-objdump -s application.elf

application.elf:      file format elf32-littlearm

Contents of section .text:
0000 04b02de5 0b08de2 28309fe5 0620a0e3 .....(0... .
0010 002083e5 00309fe5 003093e5 033083e2 .....0...0...0...
0020 10209fe5 003082e5 08309fe5 0620a0e3 .....0...0.....
0030 002083e5 ffffffea b4000000 04b02de5 .....0...0...0...
0040 00b08de2 0cd04de2 08000be5 0c109be5 .....M.....
0050 08201be5 0c301be5 033082e0 0300a0e1 .....0...0.....
0060 0d0d08be2 0d0b09de4 1eff2fe1 04b02de5 .....0...0...0...
0070 00b08de2 14d04de2 08000be5 0c109be5 .....M.....
0080 10200be5 08201be5 0c301be5 032082e0 .....0...0...0...
0090 10301be5 033082e0 0300a0e1 00d08be2 .....0...0.....
00a0 04b09de4 1eff2fe1 .....0...0...0...
Contents of section .rodata:
00a8 14000000 14000000 .....0...0...0...
Contents of section .data:
00b0 03000000 .....0...0...0...
Contents of section .comment:
0000 4743433a 2028747e 55204172 6d20456d GCC: (GNU Arm Em
0010 62656464 65642054 6f6f6c63 6861696e bedded Toolchain
0020 2031302e 332d3230 32312e31 30292031 10.3-2021.10) 1
0030 302e332e 31203230 32313038 32342028 0.3.1 20210824 (
0040 72656c65 61736529 00 release).
Contents of section .ARM.attributes:
0000 41290000 00616561 62690001 1f000000 A)...aeabi.....
0010 05345400 06020801 09011204 14011501 .4T.....
0020 17031801 19011a01 1e06 .....0...0...0...
PS C:\IFX_EMBEDDED_SYSTEMS\Baremetal_V0>

```

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

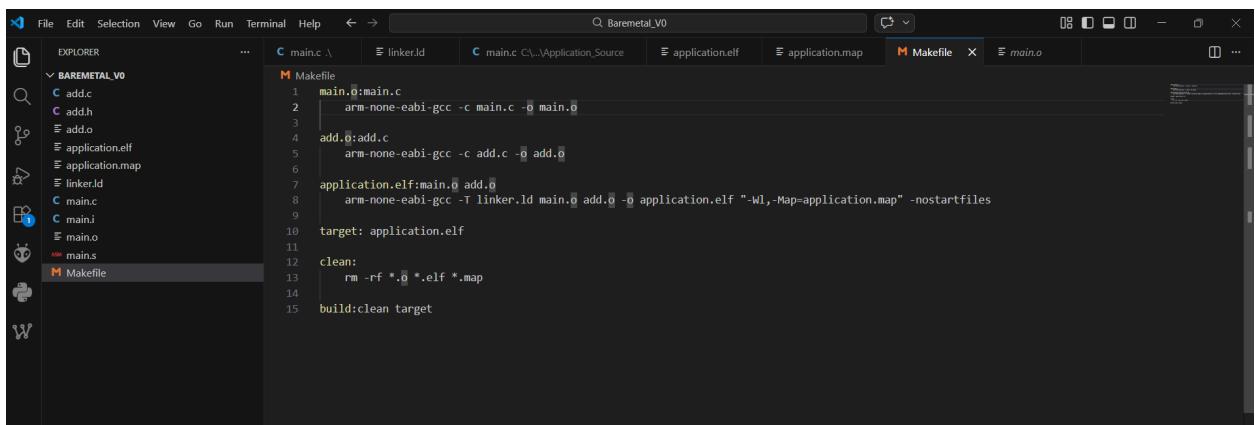
The `arm-none-eabi-objdump -s` command displays the raw byte-level contents of each section in an ELF file, allowing developers to inspect the actual machine code and initialized data that will reside in memory.

## 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



The screenshot shows a terminal window titled "BAREMETAL\_V0" with the following Makefile content:

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

add.o:add.c
    arm-none-eabi-gcc -c add.c -o add.o

application.elf:main.o add.o
    arm-none-eabi-gcc -T linker.ld main.o add.o -o application.elf "-Wl,-Map=application.map" -nostartfiles

target: application.elf

clean:
    rm -rf *.o *.elf *.map

build:clean target
```