typedef command

the C programming language provides a keyword called **typedef** to set an alternate name to an existing data type.

The **typedef** keyword in C is very useful in assigning a convenient alias to a built-in data type as well as any derived data type such as a struct, a union or a pointer.

Sometimes it becomes clumsy to use a data type with a longer name (such as "**struct structname**" or "**unsigned int**") every time a variable is declared. In such cases, we can assign a handy shortcut to make the code more readable.

typedef Syntax

In general, the **typedef** keyword is used as follows –

```
typedef existing_type new_type;
```

typedef Examples

Example 1

In C language, the keyword "**unsigned**" is used to declare unsigned integer variables that can store only non-negative values.

C also has a keyword called "**short**" that declares an integer data type that occupies 2 bytes of memory. If you want to declare a variable that is **short** and can have only non-negative values, then you can combine both these keywords (unsigned and short):

```
short unsigned int x;
```

If there are going to be many variables to be declared of this type, it will not be very convenient to use these three keywords every time. Instead, you can define an **alias** or a shortcut with the **typedef** keyword as follows –

```
typedef short unsigned int USHORT;
```

```
USHORT x;
```

Example 2

```c
#include <stdio.h>

int main() {
    typedef short unsigned int USHORT;
    typedef signed long int SLONG;

    static SLONG x = 100;
    USHORT y = 200;

    printf("Size of SLONG: %d \nSize of USHORT: %d", sizeof(SLONG), sizeof(USHORT));

    return 0;
}
```

```
Size of SLONG: 4
Size of USHORT: 2
```

Defining a Structure using Typedef

```c
#include <stdio.h>

int main() {

    typedef unsigned long int ULONG;
    typedef short int SHORT;

    struct mystruct {
        ULONG a;
        SHORT b;
    };

    typedef struct mystruct STR;
    STR s1 = {10, 20};
    printf("%ld %u", s1.a, s1.b);

    return 0;
}
```

```
10 20
```

Typedef for Struct Pointer

text.c | a.h | b.h | t.h | __register_frame_info() at 0x401288

```c
#include <stdio.h>

int main() {

    typedef unsigned long int ULONG;
    typedef short int SHORT;

    typedef struct mystruct {
        ULONG a;
        SHORT b;
    } STR;

    STR s1 = {10, 20};
    typedef STR * strptr;
    strptr ptr = &s1;

    printf("%d %d \n", s1.a, s1.b);
    printf("%d %d", ptr->a, ptr->b);

    return 0;
}
```

Console

<terminated> (exit value: 0) text.exe [C/C++ Application

```
10 20
10 20
```

Typedef for Union

text.c | a.h | b.h | t.h | __register_frame_info() at 0x401288

```c
#include <stdio.h>

int main() {

    typedef unsigned long int ULONG;
    typedef short int SHORT;

    typedef union myunion {
        char a;
        int b;
        double c;
    } UNTYPE;

    UNTYPE u1;
    u1.c = 65.50;

    typedef UNTYPE * UNPTR;
    UNPTR ptr = &u1;

    printf("a:%c b: %d c: %lf \n", u1.a, u1.b, u1.c);
    printf("a:%c b: %d c: %lf \n", ptr->a, ptr->b, ptr->c);

    return 0;
}
```

Console

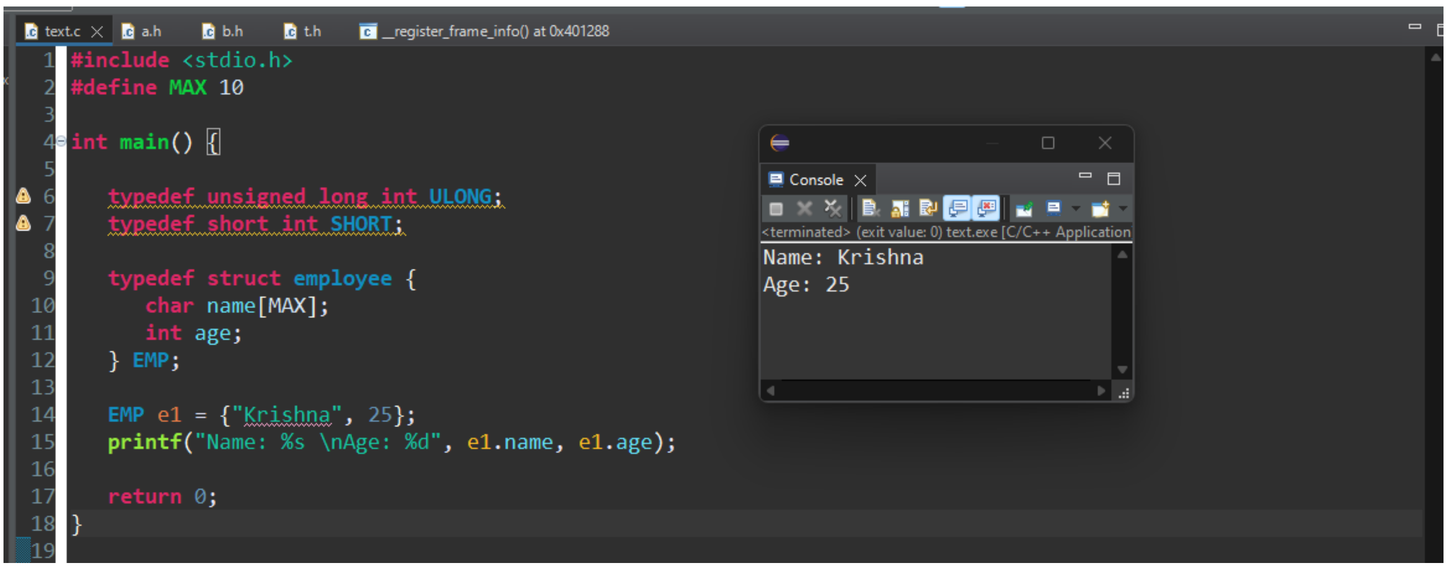
<terminated> (exit value: 0) text.exe [C/C++ Application

```
a:   b: 0 c: 65.500000
a:   b: 0 c: 65.500000
```

typedef vs #define in C

In C language, **#define** is a preprocessor directive. It is an effective method to define a constant. **#define** is a preprocessor directive, while **typedef** is evaluated at the time of compilation.

- **typedef** is limited to giving symbolic names to types only.
- **#define** can be used to define alias for values as well. For example, you can define "1" as "ONE".
- **typedef** interpretation is performed by the compiler.
- **#define** statements are processed by the pre-processor.



Header protection in c

Header protection in C is a mechanism used to prevent multiple inclusions of the same header file in a single translation unit, which can lead to errors such as redefinition of types, functions, and variables. This is typically achieved using **include guards** or **pragma once**.

Using Include Guards

Include guards are preprocessor directives that ensure the contents of a header file are included only once. Here's an example:

c نسخ الكود

```c
#ifndef HEADER_FILE_NAME_H
#define HEADER_FILE_NAME_H

// Declarations and definitions

#endif // HEADER_FILE_NAME_H
```

- #ifndef HEADER_FILE_NAME_H checks if HEADER_FILE_NAME_H is not defined.
- #define HEADER_FILE_NAME_H defines HEADER_FILE_NAME_H.
- The actual contents of the header file (declarations and definitions) are placed between the #ifndef and #endif directives.
- #endif ends the conditional preprocessor directive.

Using #pragma once

#pragma once is a preprocessor directive that serves the same purpose as include guards but is more concise. It is not part of the C standard but is supported by most modern compilers.

Here's an example:

c نسخ الكود

```c
#pragma once

// Declarations and definitions
```

Example

- **my_header.h**
- Using include guards:
- In both cases, the header file my_header.h will be included only once, preventing redefinition errors.

c نسخ الكود

```c
#ifndef MY_HEADER_H
#define MY_HEADER_H

void my_function();

#endif // MY_HEADER_H
```

Using `#pragma once`:

c نسخ الكود

```c
#pragma once

void my_function();
```

main.c

c نسخ الكود

```c
#include "my_header.h"
#include "my_header.h" // This inclusion will be ignored due to header protection

int main() {
    my_function();
    return 0;
}
```

Optimization in the context of the GNU Compiler Collection (GCC)

Optimization in the context of the GNU Compiler Collection (GCC) involves adjusting the code compilation process to improve various aspects of the generated executable, such as speed, size, and efficiency. GCC offers several optimization levels that control the degree and type of optimizations applied.

Optimization Levels in GCC

-O0: No optimization (default)

- This level disables all optimization techniques. The primary focus is on reducing the compilation time and improving the debugging experience. It preserves the original code structure as much as possible, which helps with debugging.

-O1: Basic optimization

- This level enables simple optimizations that do not significantly increase the compilation time. These optimizations improve the performance of the generated code without greatly affecting its size. Examples include removing redundant instructions and simplifying control flows.

-O2: Further optimization

- This level includes all -O1 optimizations and adds more aggressive techniques that can significantly improve the performance of the generated code. It focuses on reducing code size and execution time while ensuring that the compilation process remains reasonably fast. Common optimizations at this level include inlining of functions, vectorization, and loop unrolling.

-O3: Maximum optimization

- This level includes all -O2 optimizations and enables even more aggressive techniques that can further enhance performance. However, it may increase the size of the generated code and the compilation time. Examples of additional optimizations include aggressive function inlining and better use of vector instructions.

-Os: Optimize for size

- This level aims to reduce the size of the generated code while applying optimizations that do not significantly increase the code size. It is similar to -O2 but with a focus on minimizing the code footprint, making it ideal for embedded systems with limited memory.

-Ofast: Fastest possible code

- This level includes all -O3 optimizations and applies additional aggressive techniques that may not strictly adhere to language standards. It aims to generate the fastest possible code but can result in code that is less portable or less predictable.

-Og: Optimization for debugging

- This level is designed to offer a good balance between optimization and debugging. It enables optimizations that do not interfere with the debugging experience, making it easier to debug optimized code.

Usage

To use these optimization levels, you can pass the appropriate flag to GCC during compilation. For example:

bash نسخ الكود

```bash
gcc -O2 -o my_program my_program.c
```

This command tells GCC to compile my_program.c with the -O2 optimization level, generating an executable named my_program.

Choosing the Right Level

-O0: Use during development and debugging.

-O1: Use when you want some performance improvements without significantly increasing the compilation time.

-O2: A good balance between performance and compilation time for most production code.

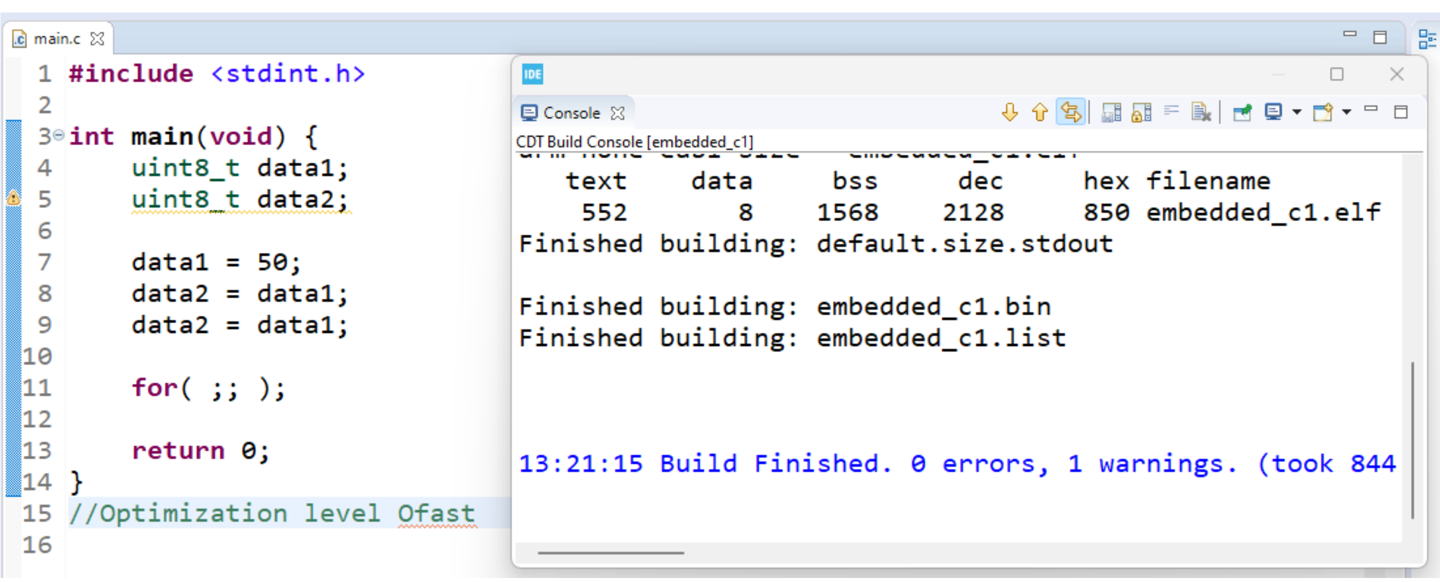
-O3: Use for compute-intensive applications where performance is critical.

-Os: Ideal for embedded systems or applications where memory is constrained.

-Ofast: Use when maximum performance is needed and strict adherence to standards is not a concern.

-Og: Use during development when you want some optimizations without sacrificing debugging capabilities.

if the code crashes and the program doesn't end or gives unexpected results. In order to understand where is the problem it's necessary to open the Disassembly section in the Debugging mode and control in the register windows how the data is trasferred to registers



```c
#include <stdint.h>

int main(void) {
    uint8_t data1;
    uint8_t data2;

    data1 = 50;
    data2 = data1;
    data2 = data1;

    for( ;; );

    return 0;
}
//Optimization level O0
```

```
   text    data     bss     dec     hex filename
    568       8    1568    2144     860 embedded_c1.elf
Finished building: default.size.stdout

Finished building: embedded_c1.bin
Finished building: embedded_c1.list


13:17:04 Build Finished. 0 errors, 1 warnings. (took 599
```

```c
#include <stdint.h>

int main(void) {
    uint8_t data1;
    uint8_t data2;

    data1 = 50;
    data2 = data1;
    data2 = data1;

    for( ;; );

    return 0;
}
//Optimization level Og
```

```
   text    data     bss     dec     hex filename
    552       8    1568    2128     850 embedded_c1.elf
Finished building: default.size.stdout

Finished building: embedded_c1.bin
Finished building: embedded_c1.list


13:17:44 Build Finished. 0 errors, 1 warnings. (took 866
```

```c
#include <stdint.h>

int main(void) {
    uint8_t data1;
    uint8_t data2;

    data1 = 50;
    data2 = data1;
    data2 = data1;

    for( ;; );

    return 0;
}
//Optimization level Os
```

```
   text    data     bss     dec     hex filename
    552       8    1568    2128     850 embedded_c1.elf
Finished building: default.size.stdout

Finished building: embedded_c1.bin
Finished building: embedded_c1.list


13:20:24 Build Finished. 0 errors, 1 warnings. (took 879
```

main.c

```c
#include <stdint.h>

int main(void) {
    uint8_t data1;
    uint8_t data2;

    data1 = 50;
    data2 = data1;
    data2 = data1;

    for( ;; );

    return 0;
}
//Optimization level O1
```

Console

CDT Build Console [embedded_c1]

```
   text    data     bss     dec     hex filename
    552       8    1568    2128     850 embedded_c1.elf
Finished building: default.size.stdout

Finished building: embedded_c1.bin
Finished building: embedded_c1.list


13:18:26 Build Finished. 0 errors, 1 warnings. (took 828
```

main.c

```c
#include <stdint.h>

int main(void) {
    uint8_t data1;
    uint8_t data2;

    data1 = 50;
    data2 = data1;
    data2 = data1;

    for( ;; );

    return 0;
}
//Optimization level O2
```

Console

CDT Build Console [embedded_c1]

```
arm-none-eabi-objdump -h -S  embedded_c1.elf  > "embedde
arm-none-eabi-objcopy  -O binary  embedded_c1.elf  "embe
arm-none-eabi-size   embedded_c1.elf
   text    data     bss     dec     hex filename
    552       8    1568    2128     850 embedded_c1.elf
Finished building: default.size.stdout

Finished building: embedded_c1.bin
Finished building: embedded_c1.list
```

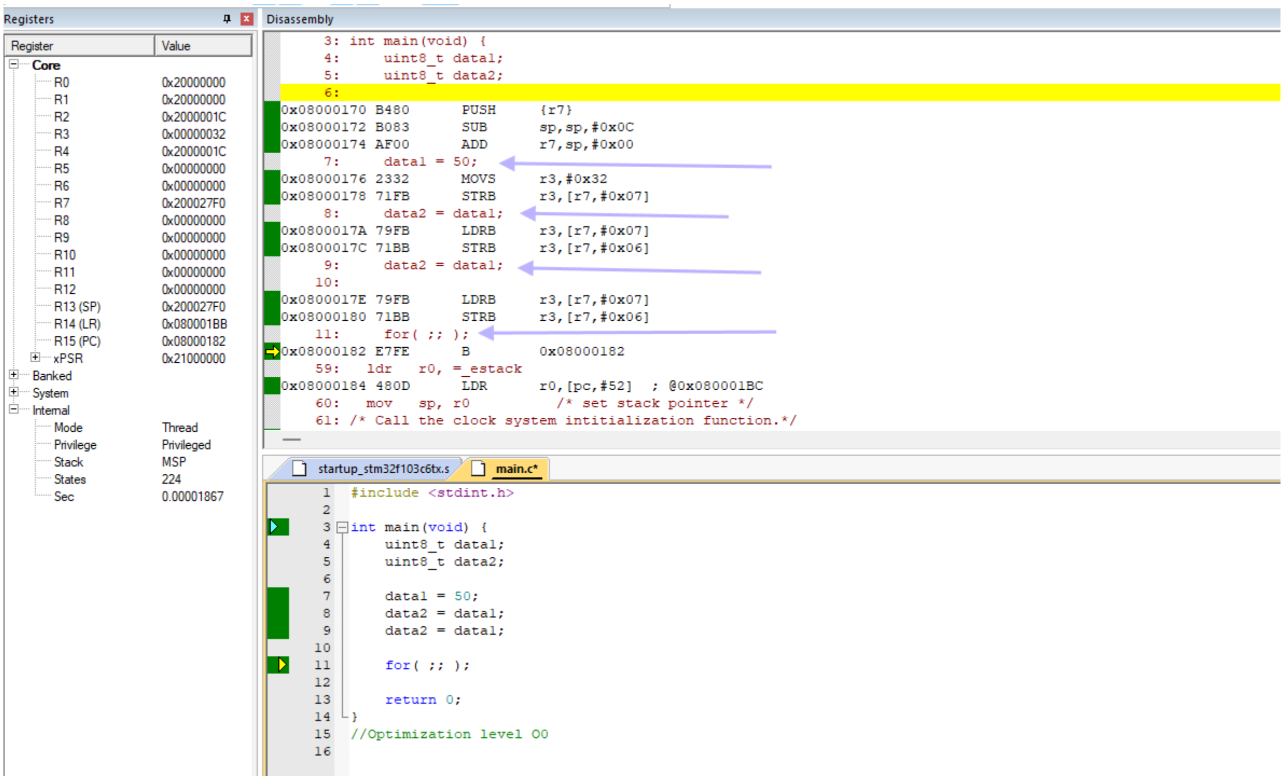
main.c

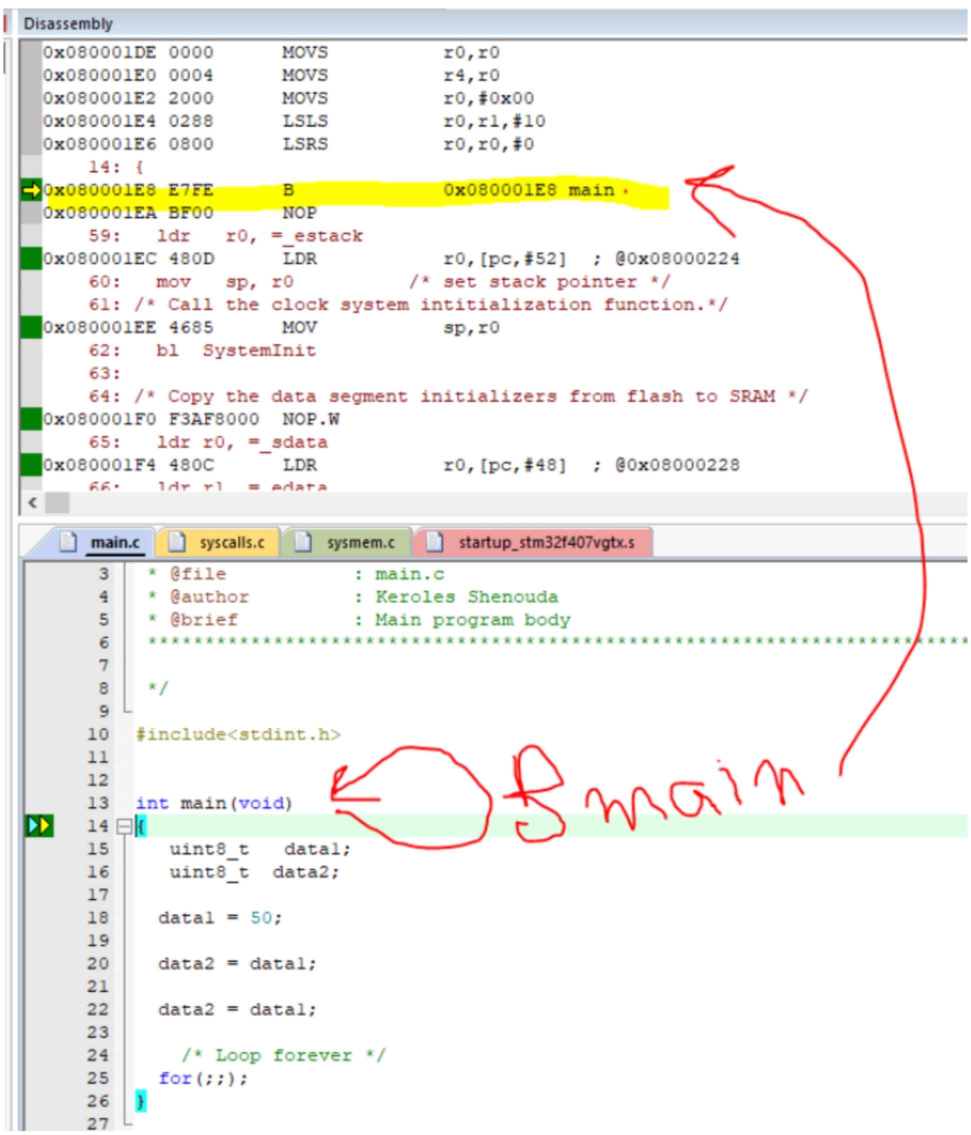
```c
embedded_c1/Src/main.c dint.h>

int main(void) {
    uint8_t data1;
    uint8_t data2;

    data1 = 50;
    data2 = data1;
    data2 = data1;

    for( ;; );

    return 0;
}
//Optimization level O3
```

Console

CDT Build Console [embedded_c1]

```
   text    data     bss     dec     hex filename
    552       8    1568    2128     850 embedded_c1.elf
Finished building: default.size.stdout

Finished building: embedded_c1.bin
Finished building: embedded_c1.list


13:19:34 Build Finished. 0 errors, 1 warnings. (took 765
```


Registers
Disassembly
Register
Value
Core
R0 0x20000000
R1 0x20000000
R2 0x2000001C
R3 0x00000032
R4 0x2000001C
R5 0x00000000
R6 0x00000000
R7 0x200027F0
R8 0x00000000
R9 0x00000000
R10 0x00000000
R11 0x00000000
R12 0x00000000
R13 (SP) 0x200027F0
R14 (LR) 0x080001BB
R15 (PC) 0x08000182
xPSR 0x21000000
Banked
System
Internal
Mode Thread
Privilege Privileged
Stack MSP
States 224
Sec 0.00001867
3: int main(void) {
4: uint8_t data1;
5: uint8_t data2;
6:
0x08000170 B480 PUSH {r7}
0x08000172 B083 SUB sp,sp,#0x0C
0x08000174 AF00 ADD r7,sp,#0x00
7: data1 = 50;
0x08000176 2332 MOVS r3,#0x32
0x08000178 71FB STRB r3,[r7,#0x07]
8: data2 = data1;
0x0800017A 79FB LDRB r3,[r7,#0x07]
0x0800017C 71BB STRB r3,[r7,#0x06]
9: data2 = data1;
10:
0x0800017E 79FB LDRB r3,[r7,#0x07]
0x08000180 71BB STRB r3,[r7,#0x06]
11: for(;;);
0x08000182 E7FE B 0x08000182
59: ldr r0, =_estack
0x08000184 480D LDR r0,[pc,#52] ; @0x080001BC
60: mov sp, r0 /* set stack pointer */
61: /* Call the clock system intitialization function.*/
startup_stm32f103c6tx.s
main.c*
1 #include <stdint.h>
2
3 int main(void) {
4 uint8_t data1;
5 uint8_t data2;
6
7 data1 = 50;
8 data2 = data1;
9 data2 = data1;
10
11 for(;;);
12
13 return 0;
14 }
15 //Optimization level O0
16


Disassembly
0x080001DE 0000 MOVS r0,r0
0x080001E0 0004 MOVS r4,r0
0x080001E2 2000 MOVS r0,#0x00
0x080001E4 0288 LSLS r0,r1,#10
0x080001E6 0800 LSRS r0,r0,#0
14: {
0x080001E8 E7FE B 0x080001E8 main
0x080001EA BF00 NOP
59: ldr r0, =_estack
0x080001EC 480D LDR r0,[pc,#52] ; @0x08000224
60: mov sp, r0 /* set stack pointer */
61: /* Call the clock system intitialization function.*/
0x080001EE 4685 MOV sp,r0
62: bl SystemInit
63:
64: /* Copy the data segment initializers from flash to SRAM */
0x080001F0 F3AF8000 NOP.W
65: ldr r0, =_sdata
0x080001F4 480C LDR r0,[pc,#48] ; @0x08000228
main.c
syscalls.c
sysmem.c
startup_stm32f407vgtx.s
3 * @file : main.c
4 * @author : Keroles Shenouda
5 * @brief : Main program body
6 **
7
8 */
9
10 #include<stdint.h>
11
12
13 int main(void)
14 {
15 uint8_t data1;
16 uint8_t data2;
17
18 data1 = 50;
19
20 data2 = data1;
21
22 data2 = data1;
23
24 /* Loop forever */
25 for(;;);
26 }
27
B main

Volatile Type Qualifier

the volatile type qualifier in C is used to inform the compiler that a variable's value may be changed at any time without any action being taken by the code the compiler finds nearby.

This prevents the compiler from optimizing code in a way that assumes the value of the variable cannot change unexpectedly.

The volatile qualifier is essential in certain scenarios, particularly in embedded systems, where hardware peripherals, interrupts, or multi-threaded programs can alter the value of a variable.

Proper Use of C's volatile Keyword

- Memory-mapped peripheral registers
- Global variables modified by an interrupt service routine
- Global variables accessed by multiple tasks within a multi-threaded application

Declaration

1. **Declaration 1:**

c　　نسخ الكود

```c
volatile uint8_t * pReg;
```

2. **Declaration 2:**

c　　نسخ الكود

```c
uint8_t volatile * pReg;
```

- uint8_t volatile * pReg;
- volatile uint8_t * pReg;
  - pointer to a volatile unsigned 8-bit integer

- int * volatile p;
  - Volatile pointers to non-volatile data

- int volatile * volatile p;
  - volatile pointer to a volatile variable

declare a pointer to a volatile unsigned 8-bit integer

1. **Declaration 1:**

```c
volatile uint8_t * pReg;
```

2. **Declaration 2:**

```c
uint8_t volatile * pReg;
```

Explanation

- **volatile uint8_t * pReg;** : This declares pReg as a pointer to a volatile uint8_t. It means that the uint8_t value that pReg points to is volatile, and therefore, it may change at any time without any action from the code.

- **uint8_t volatile * pReg;** : This declaration does the same thing. The volatile keyword can be placed after the type specifier (uint8_t) and it has the same effect as placing it before the type specifier.

When to Use volatile

Hardware Registers:

When accessing memory-mapped hardware registers in embedded systems. These registers can be changed by hardware, so the compiler should always read their values directly from memory rather than using cached values.

```c
volatile int *status_register = (int *)0x40021000;
```

Interrupt Service Routines (ISRs):

- Variables that can be modified by an interrupt service routine (ISR). The main program and the ISR may both access the variable, and the ISR can change the variable at any time.

```c
volatile int timer_flag;

void timer_ISR(void) {
    timer_flag = 1;
}

void main(void) {
    while (!timer_flag) {
        // Wait for the timer ISR to set the flag
    }
}
```

Shared Variables in Multi-threaded Programs:

- Variables shared between different threads or tasks in a multi-threaded environment. Although other synchronization mechanisms (like mutexes) are usually required to ensure safe access, volatile ensures the variable's value is not cached between accesses.

```c
volatile int shared_data;

void thread1(void) {
    shared_data = 1;
}

void thread2(void) {
    while (!shared_data) {
        // Wait for thread1 to set the data
    }
}
```

How volatile Works

When a variable is declared with the volatile qualifier, the compiler generates code that always reads the variable from memory instead of using a cached value stored in a register. This ensures the program sees the most recent value of the variable at all times.

Example

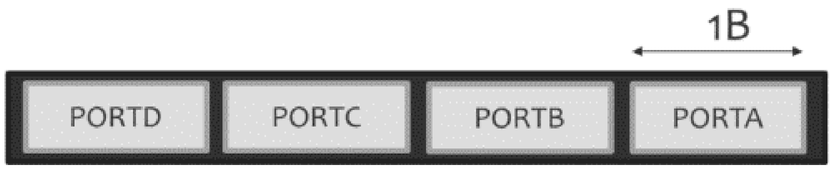
c نسخ الكود

```c
#include <stdint.h>

#define STATUS_REG   (*(volatile uint32_t *)0x40021000)

void check_status(void) {
    while (STATUS_REG != 0) {
        // Perform some action based on the status register
    }
}
```

In this example, **STATUS_REG** is a memory-mapped hardware register. By declaring it as volatile, the compiler is instructed to always read its value from the specified memory address, ensuring the latest value is used each time it is accessed.

Now let's see how access the register absolute address

Write 0xFFFFFFFF on SIU register which have absolute address 0x30610000




```c
#include <stdint.h>

typedef union {
    uint32_t ALL_ports;
    struct {
        uint32_t PORTA:8 ;
        uint32_t PORTB:8 ;
        uint32_t PORTC:8 ;
        uint32_t PORTD:8 ;
    } SIU_fields;
} SIU_R;

#define SIU_REGISTER_ADDRESS 0x306100

int main(void) {
    // Define a pointer to the SIU register
    volatile SIU_R* PORTS = (volatile SIU_R*) SIU_REGISTER_ADDRESS;

    // Set all bits of the register to 0xFFFFFFFF
    PORTS->ALL_ports = 0xFFFFFFFF;

    // Set only PORTA to 0xFF, other fields remain unchanged
    PORTS->SIU_fields.PORTA = 0xFF;

    // Your code here

    return 0;
}
```

Solution 2

```c
*((volatile unsigned long*)(0x306100)) = 0xFFFFFFFF;
```

Or

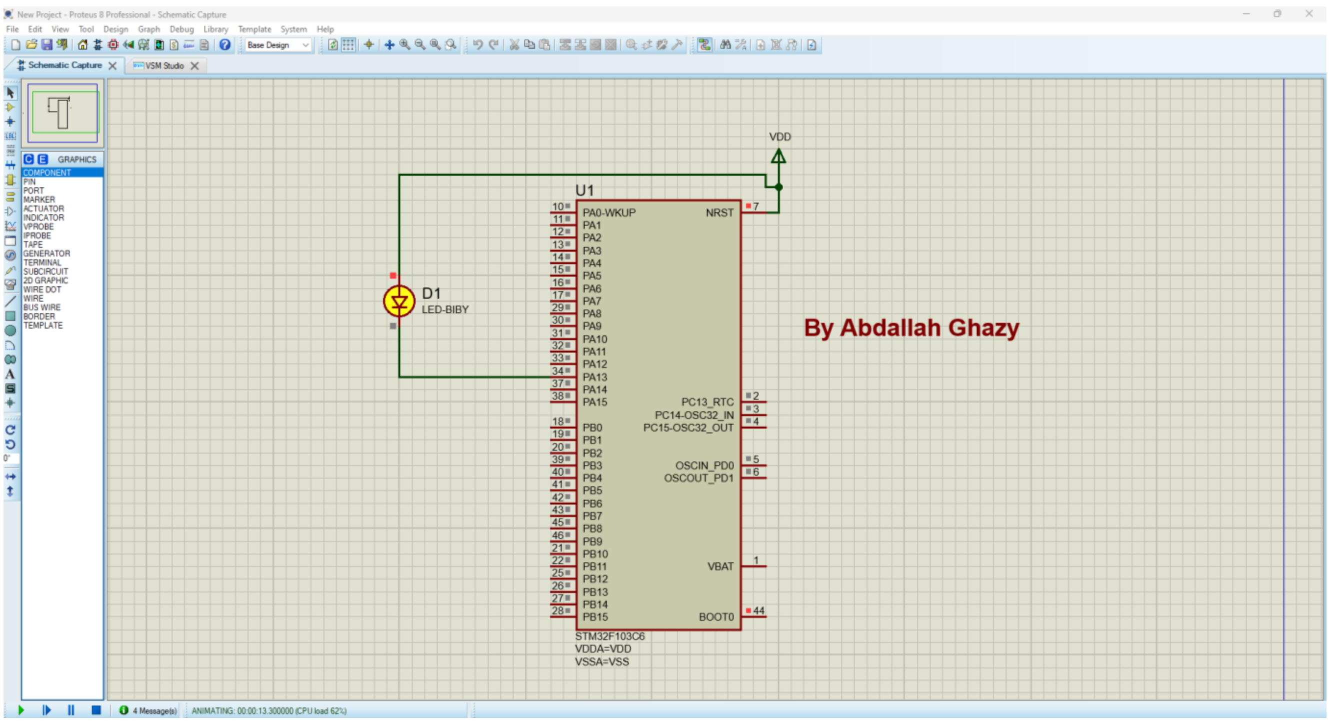
```c
#define MYREGISTER *((volatile unsigned long*)(0x306100))
```

Toole led on Stm32f103CX

Led is connected to GPIO port A13

To make a GPIO toggling in STM32, you need to work with two peripherals:

- RCC (reset and clock control)
    - The RCC is necessary because the GPIO has disabled clock by default
- GPIOx (general purpose input/output).



```c
#include <stdint.h>

typedef volatile unsigned int vint32;

#define HIGH 1
#define LOW  0

#define RCC_BASE       0x40021000
#define PORTA_BASE     0x40010800

#define RCC_APB2ENR    *((volatile uint32_t*)(RCC_BASE+0x18))
#define GPIOA_CRH      *((volatile uint32_t*)(PORTA_BASE+0x04))
//#define GPIOA_ODR    *((volatile uint32_t*)(PORTA_BASE+0x0C))
#define GPIOA_ODR       (PORTA_BASE + 0x0C)

#define RCC_APB2ENR_IOPAEN (1<<2)

typedef union {
	vint32 allFields;

	struct {
		vint32 :13;
		vint32 pin13 :1;
	}pin;

} R_ODR_t;

int main(void) {

	volatile R_ODR_t *R_ODR =  (volatile R_ODR_t*)GPIOA_ODR;

	RCC_APB2ENR |= RCC_APB2ENR_IOPAEN;
	GPIOA_CRH &= 0xff0fffff;
	GPIOA_CRH |= 0x00200000;

	while (1) {

		//GPIOA_ODR |= 1 << 13;
		R_ODR->pin.pin13 = HIGH;
		for (int i = 0; i < 5000; i++);

		//GPIOA_ODR &= ~(1 << 13);
		R_ODR->pin.pin13 = LOW;
		for (int i = 0; i < 5000; i++);
	}

	return 0;
}
```

Understanding const and volatile Together

Qualifiers Explanation:

- **const**: Indicates that the value of the variable should not be changed by the program. The compiler will enforce this restriction, meaning you cannot modify the value through that pointer or reference.

- **volatile**: Indicates that the value of the variable can change at any time without any action being taken by the code the compiler finds nearby. This is used to tell the compiler not to optimize accesses to this variable, as it might be changed by hardware or other threads.

Pointer Definitions:

- **uint32_t volatile * const Preg1**:
  - Preg1 is a constant pointer to a volatile uint32_t.
  - This means that the address stored in Preg1 cannot be changed (i.e., Preg1 itself is constant), but the value at that address can be changed unexpectedly (e.g., by hardware).

- Example:

c نسخ الكود

```c
uint32_t volatile * const Preg1 = (uint32_t volatile *)0xFFFF0000;
// You can read from or write to *Preg1, but you can't change Preg1 itself.
```

uint32_t const volatile * const Preg2:

- Preg2 is a constant pointer to a const volatile uint32_t.
- This means that the address stored in Preg2 cannot be changed (i.e., Preg2 itself is constant), and the value at that address is volatile but also constant (read-only) in the context of the program.

- Example:

c نسخ الكود

```c
uint32_t const volatile * const Preg2 = (uint32_t const volatile *)0xFFFF0004;
// You can only read from *Preg2. The address Preg2 itself can't be changed.
```

Practical Implications

- **const volatile**: This combination is often used for hardware registers that are read-only but may be updated by hardware. It ensures that the compiler does not optimize away accesses to the register because the hardware might change its value. At the same time, it prevents the programmer from modifying the register's value directly.

Summary

- **const** prevents modification of the pointer itself.
- **volatile** prevents optimization of the memory accesses to handle unexpected changes.
- **const volatile** indicates that the data is read-only from the perspective of the programmer, but it may be updated by external factors (e.g., hardware).

Usage Context

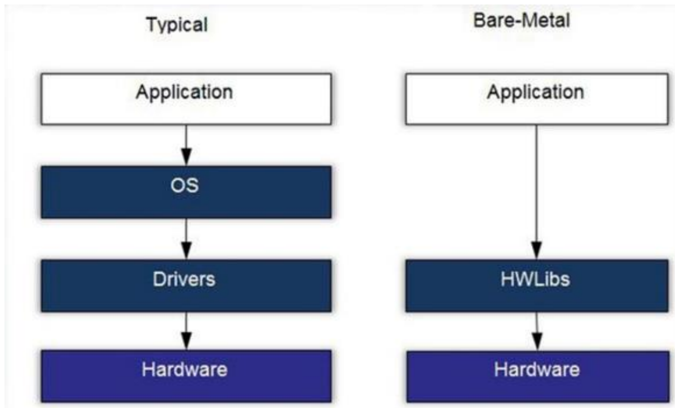
- **Preg1**:
  - **Example**: A hardware register that can be written to and read from, but the address of the register should not be changed.
  - **Use Case**: Register configuration where you configure something once and monitor the status.

- **Preg2**:
  - **Example**: A read-only status register that can be changed by hardware but should not be modified by the programmer.
  - **Use Case**: Reading status flags or error codes from hardware peripherals.

Bare metal Embedded SW

are metal embedded software referring to programming in embedded systems without an operating system (OS) or middleware layers.

This approach involves writing low-level code that interacts directly with the hardware, often in resource-constrained environments.



tool-chain

A **toolchain** in embedded systems development is a set of programming tools used to develop, compile, and debug software for embedded devices.

It typically includes a compiler, assembler, linker, and debugger, along with other utilities.

Native Toolchain

Native toolchain refers to a set of development tools used to compile and build software for the same architecture and operating system on which the development is performed. In other words, the toolchain is designed for the host machine's architecture.

Components:

- **Compiler:** Compiles source code into machine code for the host system. For example, gcc for Linux or cl for Windows.
- **Assembler:** Converts assembly language code into machine code.
- **Linker:** Links object files into executables or libraries for the host system.
- **Debugger:** Debugs the application running on the host system.
- **Libraries:** Standard libraries and runtime for the host system.

Example Use Case:

- Developing applications on a Linux PC that runs on an x86 architecture. The toolchain will compile code for the same x86 architecture and Linux OS.

Example Toolchain:

- ***GCC (GNU Compiler Collection):*** Provides a native compiler for various operating systems and architectures.

bash نسخ الكود

```bash
gcc -o myprogram myprogram.c
```

Cross Compiling Toolchain

Cross-compiling toolchain is used to compile code on a host system for a target system with a different architecture or operating system. The toolchain generates binaries for a target platform different from the one used for development.

Components:

- ***Cross-Compiler:*** A compiler that generates code for the target architecture. For example, arm-none-eabi-gcc for ARM Cortex-M microcontrollers.
- ***Cross-Assembler:*** Assembles code for the target architecture.
- ***Cross-Linker:*** Links object files to create executables for the target system.
- ***Cross-Debugger:*** Debugs applications running on the target system. For example, gdb with a remote connection.

Example Use Case:

- Developing firmware for an ARM microcontroller on a Linux PC. The cross-compiling toolchain will generate code for the ARM architecture, not x86.

Example Toolchain:

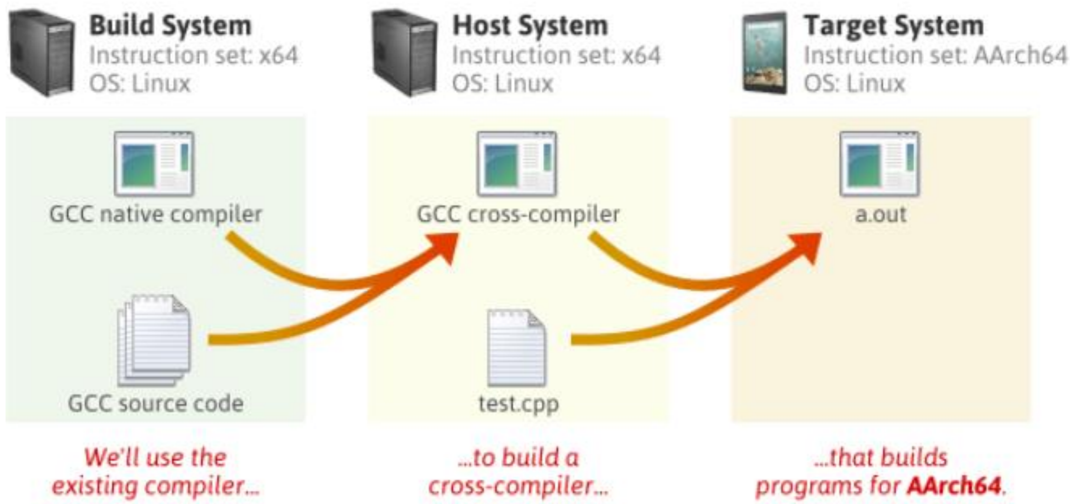
- ***GNU Arm Embedded Toolchain:*** A cross-compiling toolchain for ARM Cortex-M and Cortex-R processors.

bash نسخ الكود

```bash
arm-none-eabi-gcc -mcpu=cortex-m4 -mthumb -o myprogram.elf myprogram.c
```

Definition

- The build machine, where the toolchain is built.
- The host machine, where the toolchain will be executed.
- The target machine, where the binaries created by the toolchain are executed



Components – GCC

GCC: The GNU Compiler Collection (Table Summary)

Feature	Description	Example
Basic Compilation	Compiles source code (e.g., main.c) into an executable by default named a.out.	gcc main.c
Output Name	Specifies the name of the generated executable file.	gcc main.c -o test (executable named test)
Include Paths	Tells GCC where to search for header files used by the source code.	gcc main.c -I/usr/share/include -I. -I./inc/ -I../../inc
Enabling Warnings	Enables all compiler warnings during the compilation process.	gcc -Wall main.c -o test
Warnings as Errors	Treats all warnings as errors, causing compilation to fail unless warnings are fixed.	gcc -Wall -Werror main.c -o test
Options File	Allows specifying compilation options in a separate text file.	gcc main.c @options-file (options in options-file)

Creating a Static Library (ar Command)

Command (Options)	Description	Example
ar [rcs] library object_files	Creates a new static library (library) or adds object files (object_files) to an existing one.	ar rcs libmylib.a file1.o file2.o (creates libmylib.a from file1.o and file2.o)
ar r library object_files	Adds object files (object_files) to an existing static library (library).	ar r libmylib.a file3.o (adds file3.o to libmylib.a)
ar c library object_files	Creates new members (object files) in an existing static library (library).	(Same as ar r)
ar d library object_files	Deletes members (object files) from a static library (library).	ar d libmylib.a file3.o (removes file3.o from libmylib.a)
ar t library	Displays a table of contents for a static library (library), listing member names.	ar t libmylib.a (shows object files in libmylib.a)
ar x library	Extracts members (object files) from a static library (library) into the current directory.	ar x libmylib.a (extracts all files from libmylib.a)

Key Points:

- r: Creates a new archive or replaces existing members.
- c: Creates new members in an existing archive (same as r).
- s: Creates an archive with symbol table information (often used with r).
- d: Deletes members from an existing archive.
- t: Displays a table of contents for the archive.
- x: Extracts members from the archive.

arm-none-eabi-gcc vs. arm-linux-gnuapi

Feature	arm-none-eabi-gcc	arm-linux-gnuapi
Target Device Type	Microcontrollers without an OS	ARM devices running Linux
Binary Interface	EABI (for bare-metal systems)	Linux-specific ABI
Operating Environment	Bare-metal (no OS)	Linux operating system
Use Cases	Embedded devices, industrial control, medical	Devices like Raspberry Pi, BeagleBone, etc.
OS Dependency	No	Yes
Example Command	`./arm-none-eabi-gcc -o main.elf main.c`	`arm-linux-gnuapi-gcc -o main main.c`
Used for Bare-metal Applications	Yes	No

the meanings of each part of the texts you mentioned:

arm-none-eabi-gcc

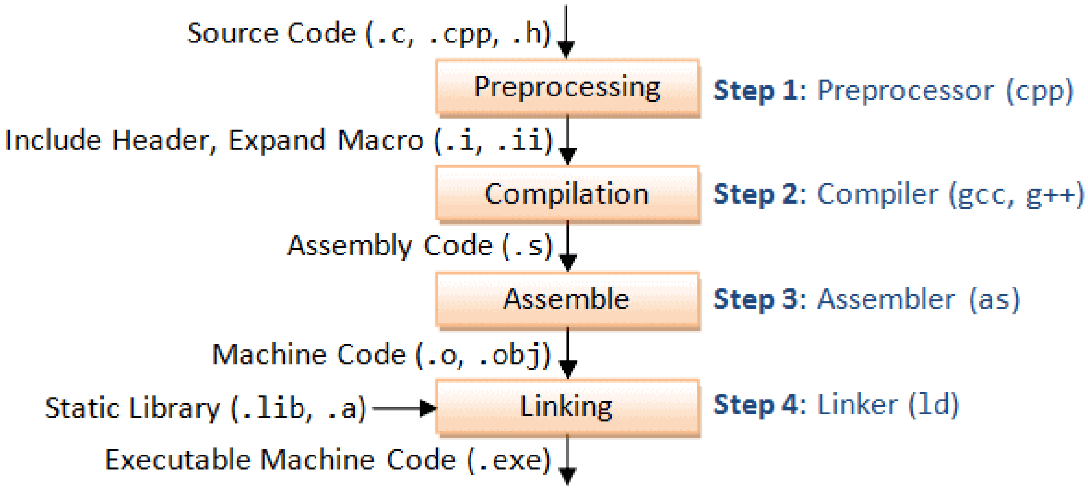
- **arm**: Indicates that the compiler is targeted for ARM architecture processors.
- **none**: Signifies that the compiler is intended for bare-metal development, meaning software runs directly on hardware without an operating system.
- **eabi**: Stands for **Embedded Application Binary Interface**, a standard for organizing software on embedded systems to coordinate code and data.
- **gcc**: Stands for GNU Compiler Collection, a suite of compilers supporting various programming languages.

arm-linux-gnuapi

- **arm**: Refers to ARM architecture processors.
- **linux**: Indicates that the compiler is targeted for software running on the Linux operating system.
- **gnu**: Refers to tools that follow the GNU Project, such as compilers and libraries.
- **api**: Stands for **Application Programming Interface**, though in this context, it might be an attempt to indicate GNU-specific API standards. The correct term usually is "gnueabi" rather than "gnuapi."

In summary, arm-none-eabi-gcc is a compiler for developing software for ARM processors in a bare-metal environment, while arm-linux-gnuapi appears to be a somewhat incorrect or incomplete reference to tools for developing software for ARM processors running Linux with GNU tools.

Compilation Process



Preprocessing:

- The preprocessor handles directives like #include, #define, and conditional compilation.
- It expands macros, includes the contents of header files, and processes conditional compilation instructions.
- The output is a preprocessed source file with the .i extension (in C) or .ii extension (in C++).

Example command:

bash نسخ الكود

```bash
gcc -E source.c -o source.i
```

Compilation:

- The compiler translates the preprocessed source code into assembly code for the target architecture.
- Syntax and semantic analysis are performed during this stage.
- The output is an assembly file with the .s extension.

Example command:

bash نسخ الكود

```bash
gcc -S source.i -o source.s
```

Assembly:

- The assembler converts the assembly code into machine code, generating an object file.
- The output is an object file with the .o (or .obj on Windows) extension.

Example command:

bash نسخ الكود

```bash
gcc -c source.s -o source.o
```

Relocatable object files

contain processor architecture-specific machine code with no absolute addresses, allowing for flexible placement in memory during the linking process. They enable the generation of executable files by allowing the linker to adjust addresses and resolve symbols according to the final memory layout of the program.

Linking:

- The linker combines object files and libraries into a single executable.
- It resolves symbol references, assigning addresses to various code and data sections.
- The output is an executable file.

Example command:

bash نسخ الكود

```bash
gcc source.o -o executable
```

Additional Details

- **Intermediate Files**: During the compilation process, various intermediate files are generated:
  - Preprocessed source file (.i or .ii).
  - Assembly file (.s).
  - Object file (.o or .obj).

- **Linking Libraries**: The linker can link against standard libraries (like the C standard library) and custom libraries.
  - Static libraries have the .a (Unix/Linux) or .lib (Windows) extension.
  - Dynamic libraries have the .so (Unix/Linux) or .dll (Windows) extension.

Compiler Flags

Compiler Flags: Various flags can be used to control the compilation process:

- -I to specify include directories for header files.
- -L to specify library directories.
- -D to define macros.
- -O to control optimization levels.
- -g to include debugging information.
- -Wall to enable all compiler's warning messages.

table of key gcc flags :

Flag	Description	Result
`-E`	Runs only the preprocessing stage.	Produces a file with the source code after processing preprocessor directives.
`-S`	Runs the compilation stage to generate assembly code.	Produces a file with the assembly code (e.g., `.s` file).
`-c`	Runs preprocessing, compilation, and assembly, but not linking.	Produces an object file (e.g., `.o` or `.obj` file).
`-o`	Specifies the name of the output file.	Determines the name of the file where the compiled code will be stored.
`-l`	Links against specified libraries.	Links the program with the specified library (e.g., `-lm` for the math library).
`-I`	Adds directories to the search path for header files.	Allows the compiler to find header files in the specified directories.
`-L`	Adds directories to the search path for libraries.	Allows the linker to find libraries in the specified directories.
`-D`	Defines macros for the preprocessor.	Defines a macro for the compiler as if it were specified in the source code (e.g., `-DDEBUG`).
`-g`	Includes debugging information in the output file.	Facilitates debugging with tools like `gdb`.
`-O`	Sets the optimization level for the compiler.	Controls the level of optimizations applied to the compiled code (e.g., `-O2`, `-O3`).
`-Wall`	Enables most compiler warning messages.	Helps identify potential issues and warnings in the code.
`-Werror`	Treats warnings as errors.	Causes the compilation to fail if there are warnings.

Compile Time Binding

- is a concept related to how addresses are assigned to code and data during the compilation process.
- if memory location is fixed and known at compile time, absolute code with absolute address can be generated.
- Must Recompile Code if starting location changed.

The. map

The .map file gives a complete listing of all code and data addresses for the final software image.

It provides information similar to the contents of the linker script described earlier.

However, these are results rather than instructions and therefore include the actual lengths of the sections and the names and locations of the public symbols found in the relocatable program.

Link with `.map` File Generation:

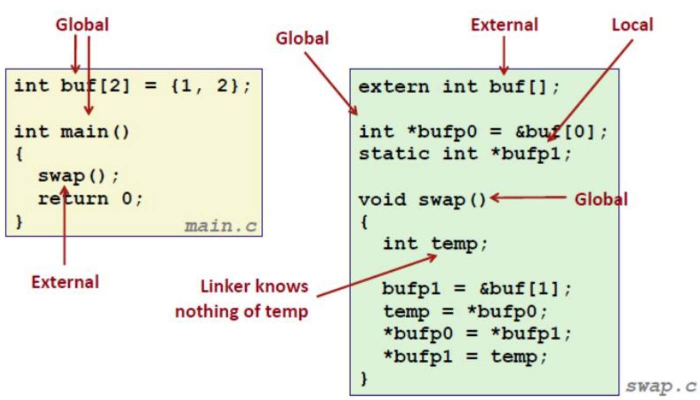
When linking the object files, use the `-Map` option to create a `.map` file. In this example, we will produce an executable file named `main.elf` and a `.map` file named `main.map`:

bash نسخ الكود

```bash
./arm-none-eabi-gcc main.o -o main.elf -Wl,-Map=main.map
```

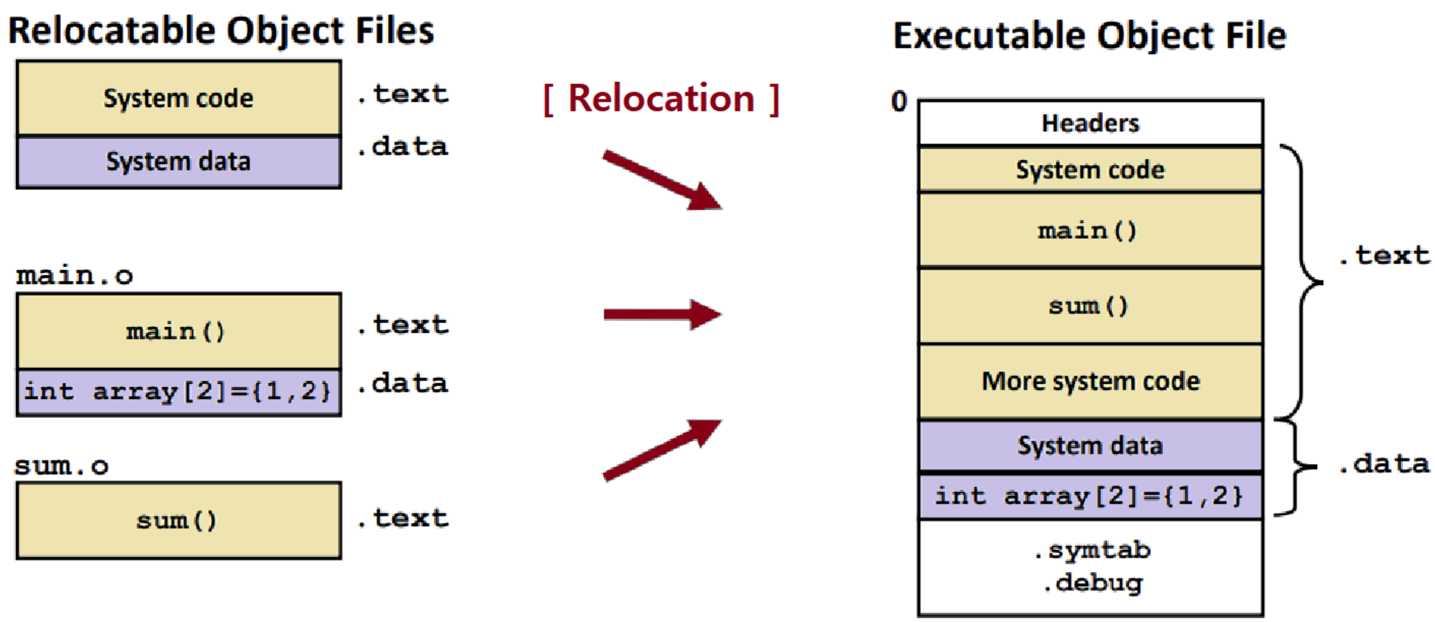
Resolving Symbols During Linking

Resolving Symbols During Linking is the process where the linker connects symbolic references in object files to their actual addresses in the final executable. This involves reading symbol tables, matching references to definitions, and adjusting addresses. This step ensures that all code and data references are correctly linked and that the final program is executable.



Relocating Code and Data During Linking

Relocating Code and Data During Linking involves adjusting addresses in object files so that they correctly fit into the final memory layout of an executable or shared library. This process includes combining object files, assigning final memory addresses, updating references based on relocation entries, and producing the final output with all symbols correctly resolved.



Tool	Purpose	Example Usage
gcc (GNU C Compiler)	Compiles C/C++ (and other languages) source code into object files and links them into executables.	gcc -o myprogram myprogram.c
ld (Linker)	Links object files generated by compilers into executable programs.	ld -o myprogram myprogram.o
make	Reads Makefiles to automate the compilation, linking, and building process of software projects.	make
ar (Archiver)	Creates and manages static libraries by archiving multiple object files into a single library file.	ar rcs libmylib.a file1.o file2.o
readelf	Analyzes and displays information about ELF format executable and object files.	readelf -a myprogram
objdump	Disassembles machine code instructions within ELF format object and executable files.	objdump -d myprogram
nm	Lists symbols (functions, variables) defined within object and executable files.	nm myprogram
strings	Extracts printable strings embedded within binary files.	strings myprogram
strip	Removes unnecessary sections from executables to reduce their file size.	strip myprogram
addr2line	Converts memory addresses from a running program back to the original source code line number.	addr2line -e myprogram 0x08000400
size	Displays the size of each section within an ELF format object or executable file, and the total file size.	size myprogram
gdb (GNU Debugger)	Powerful interactive debugger for analyzing program behavior, setting breakpoints, and inspecting variables.	gdb myprogram
cp (copy)	Copies files and directories from one location to another.	cp /source/file /destination/file
mv (move)	Moves or renames files and directories.	mv /source/file /destination/file
rm (remove)	Deletes files and directories (use with caution!).	rm /path/to/file (**Caution: Use with care!**)

mkdir (make directory)	Creates a new directory.	mkdir newdirectory
rmdir (remove directory)	Removes an empty directory.	rmdir emptydirectory
ls (list)	Lists the contents of a directory. ls -l provides a detailed listing with permissions, owner, and group.	ls or ls -l /path/to/directory
cd (change directory)	Changes the current working directory.	cd /path/to/directory
pwd (print working directory)	Prints the full path of the current working directory.	pwd
cat (concatenate)	Reads the contents of one or more files and displays them on the terminal.	cat /path/to/file
grep (global search regular expression)	Searches for lines in one or more files that match a specified pattern.	grep 'pattern' /path/to/file
less	Reads a file one page at a time, useful for viewing long files.	less /path/to/file
head	Displays the first few lines of a file.	head /path/to/file
tail	Displays the last few lines of a file.	tail /path/to/file
chmod (change mode)	Modifies file permissions to control read, write, and execute access for users, groups, and others.	chmod +x /path/to/file (makes file executable)