CSE221: Introduction to Embedded Systems

Lab #2

BY:

Omar Mosaad Atalla Abdellatif Diab

ID: 20p3176

Group 1 /Section 2

Submitted to:

Eng Hesham Salah

Task 1:

Code: -

in “types.h” file >>>

```c
// Functions >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
#define x 10

// macro function
#define ADD_MACRO(a, b) ((a) + (b))

// Function to add two variables and return the sum
int add_function(int a, int b) {
    return a + b;
}
//>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
```

In Main>>>

```c
#include <stdio.h>
#include "types.h"

int y = 5;

int main(){
    int a =5;
    int b = 3;

    int result_macro = ADD_MACRO(x, y);
    printf("Using macro: %d + %d = %d\n", x, y, result_macro);

    int result_function = add_function(x, y);
    printf("Using function: %d + %d = %d\n", x, y, result_function);

    return 0;
}
```

Output: -

```
PS C:\Users\omarmosaad\VsCode gcc lab2.c -o lab2
PS C:\Users\omarmosaad\VsCode Projects\Z____LABS\Embedded\Lab_2> ./lab2
Using macro: 10 + 5 = 15
Using function: 10 + 5 = 15
```

Answer of point 1:

1. **Resolution in build process:**
   a) Macros are resolved during **preprocessing**,
   b) Functions are resolved during **compilation**.
2. **Handling wrong parameters:**
   a) Macros **lack** type-checking, resulting in potential runtime errors.
   b) Functions **have** type-checking and produce compilation errors for wrong parameters.
3. **Sequence in function call:**
   a) Functions execute in sequence, creating stack frames.
   b) macros are expanded where invoked without explicit execution order.
4. **Macro-like functions:**

   Some languages have constructs like macros but with type-checking, scoping rules, and expanded during compilation.

Answer of point 2:

When a variable is declared in a **.h (header) file** and that file is included in **2 .c (source code) files**, it leads to **multiple definitions of the variable**.
To resolve this, we added the **extern keyword** to the variable declaration in the **.h file**. This informs the compiler that the variable is defined elsewhere and **prevents linker errors.** The actual definition of the variable should be present in **one of the .c files**, ensuring only one instance of the variable at runtime.

Task 2:

Types of building errors: -

1. Compiler Errors:

- Undeclared Variable:

```c
int result = x + 5;
```

Error: Using an undeclared variable 'x'.

- Missing Header File:

```c
#include <missing_header.h>
```

```

Error: Attempting to include a header file that does not exist.

- Incorrect Function Call:

```c
int result = add(5, 10);
```

Error: Calling a function 'add' that is not defined or declared.

- Mismatched Function Arguments:

```c
int multiply(int a, int b);
int result = multiply(5);
```

Error: Mismatch in the number of arguments for the function 'multiply'.

2. Linker Errors:

- Undefined Symbol:

```c
extern int x;
int result = x + 5;
```

Error: Using an externally declared variable 'x' without its definition.

- Circular Dependency:

```
// file1.c
extern int x;
int result = x + 5;
// file2.c
extern int result;
```
```

*int x = result * 2;*

Error: Creating a circular dependency between two variables.

3. Runtime Errors:

- Null Pointer Assignment:

int ptr = NULL;*

**ptr = 10;*

Error: Assigning a value to a null pointer.

- Accessing Freed Memory:

int ptr = malloc(sizeof(int));*

free(ptr);

**ptr = 10;*

Error: Accessing memory that has already been freed.

- Infinite Loop:

```
while (1) {
    // Code without a loop termination condition
}
```

Error: Creating an infinite loop that never terminates.

- Buffer Overflow:

```c
char buffer[5];
strcpy(buffer, "This is a longer string");
```

Error: Writing more data into a buffer than its allocated size.