

Embedded C Programming

by:
Eng. Mohamed Tarek.



- C Programming Language concepts like data types, Operators, if-else statements, loops, functions, arrays, pointers and structures.
- Basic concepts of embedded systems and microcontrollers.



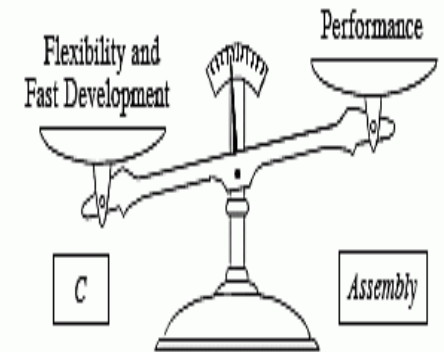
- Embedded systems are programmed using different types of languages:
 - Machine Code
 - Low level language like assembly.
 - High level language like C, C++, Java, Ada, etc.

■ Assembly Language characteristics:

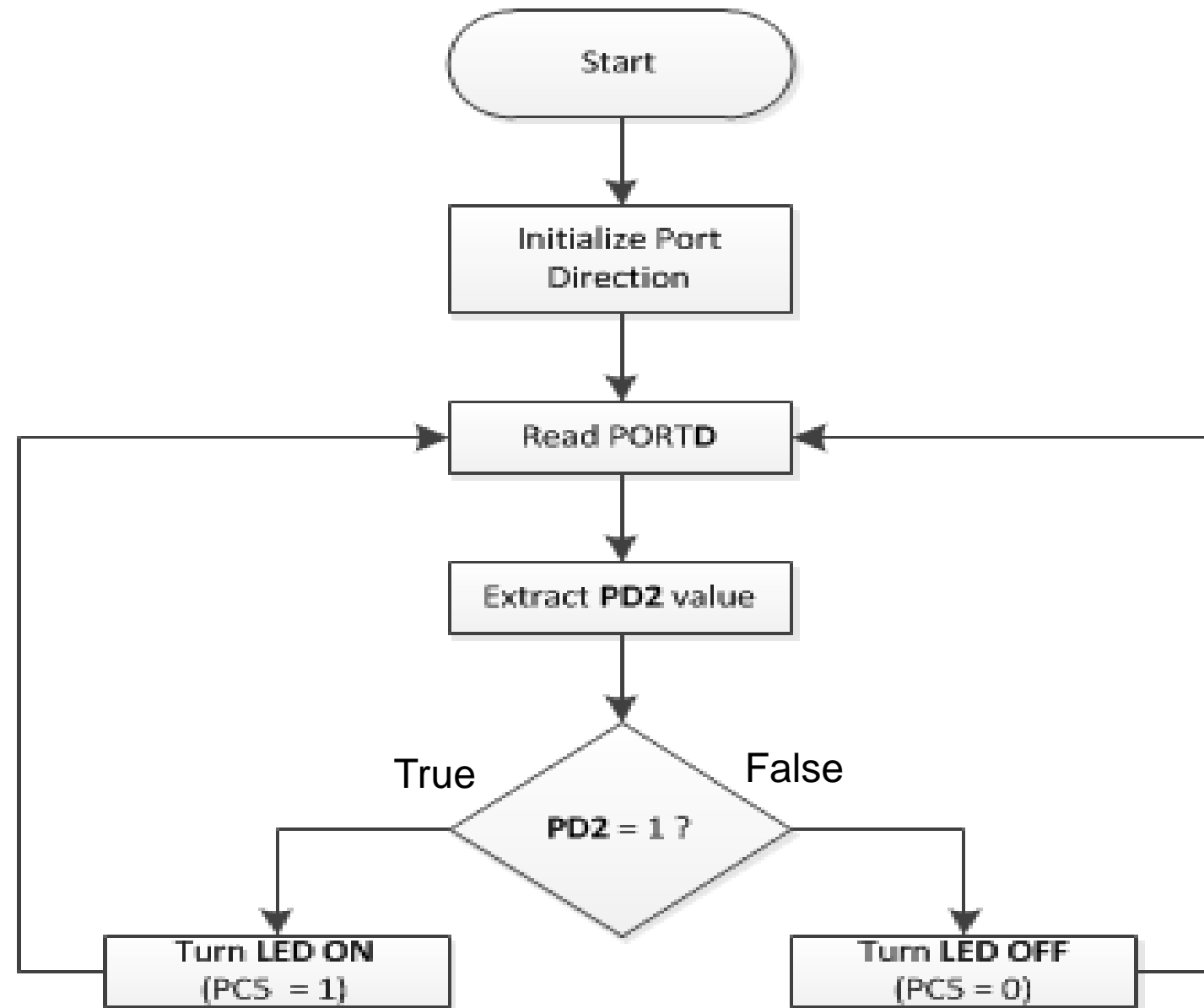
- Platform specific so code portability is not there.
- Optimized code size and speed.
- Too difficult to write large applications and consume large development time.
- Assembly codes lead to higher software development costs.
- Can be directly converted to binary machine codes that the processor uses to execute the instructions without compilation process using Assembler.

FIGURE 28-10

Assembly versus C. Programs in C are more flexible and quicker to develop. In comparison, programs in assembly often have better performance; they run faster and use less memory, resulting in lower cost.



- Write an assembly program to read the push button pin (PD2) in PORTD and according to its status (**pressed** or **released**) it will turn **ON** or **OFF** the LED connected to pin PC5 in PORTC.
- Refer to AVR Instruction Set.pdf



```
.include "m16def.inc"                                ; Header file for the ATmega16
                                                    ; microcontroller

                LDI    R16, 0x00                      ; Load 0b00000000 in R16
                OUT    DDRD, R16                      ; Configure pin PD2 as Input
                LDI    R16, 0x20                      ; Load 0b00100000 in R16
                OUT    DDRC, R16                      ; Configure pin PC5 as Output

Again:          IN     R16, PIND                      ; Read the values on the pins of PortD
                                                    ; and store in R16

                ANDI    R16, 0x04
                CPI     R16, 0x04
                BREQ    LEDON
                LDI     R16, 0x00
                OUT     PORTC, R16                    ; Write 0 to PC5 to Turn OFF the LED
                RJMP    Again

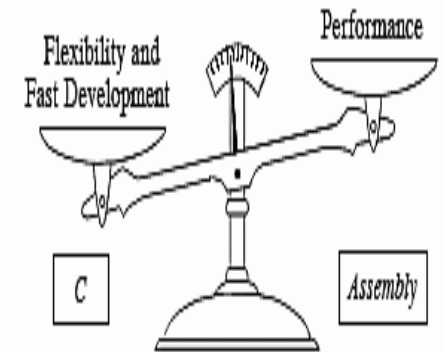
LEDON:          LDI     R16, 0x20
                OUT     PORTC, R16                    ; Write 1 to PC5 to Turn ON the LED
                RJMP    Again
```

■ C Language characteristics:

- C has advantage of target independence and is not specific to any particular platform.
- Type checking makes the program less prone to error.
- The development cycle is short.
- C codes can be developed more quickly, codes written in C offers better productivity.
- Programs developed in C are much easier to understand, maintain and debug.

FIGURE 28-10

Assembly versus C. Programs in C are more flexible and quicker to develop. In comparison, programs in assembly often have better performance; they run faster and use less memory, resulting in lower cost.



- Easier in writing large applications compared to assembly.
- It is easier to write good code in C & convert it to an efficient assembly code (using high quality compilers) rather than writing an efficient code in assembly itself.
- Benefits of assembly language programming over C are negligible when we compare the ease with which C programs are developed by programmers.

- As C combines functionality of assembly language and features of high level languages, C is treated as a ‘middle-level computer language’ or ‘high level assembly language’.

“Also, you can include in-line assembly within your C code, and thereby get the best of both worlds”

- C for Embedded Applications is a proper subset from the C Language suitable for Embedded Systems.
- Embedded Systems Programming are not like general purpose computers since we have limited hardware resources.
- Embedded C is a C with limited resources. You have to care about the memory size.
- Embedded C deal directly with HW.
- Manipulating large data structures and using recursive functions results in using large amounts of RAM and they are the techniques which are generally avoided in microcontrollers programming.

- Embedded systems programming is different from developing applications on a desktop computers:
 - Embedded devices have resource constraints (limited ROM, limited RAM, limited stack space and less processing power).
 - Components used in embedded system and PCs are different, embedded systems typically uses smaller, less power consuming components.
 - Embedded systems are more tied to the hardware.

- Two salient features of Embedded Programming are code speed and code size:
 - **Code speed** is governed by the processing power and timing constraints.
 - **Code size** is governed by available program memory and use of programming language.
- Goal of embedded system programming is to get maximum features in **minimum space** and **minimum time**.

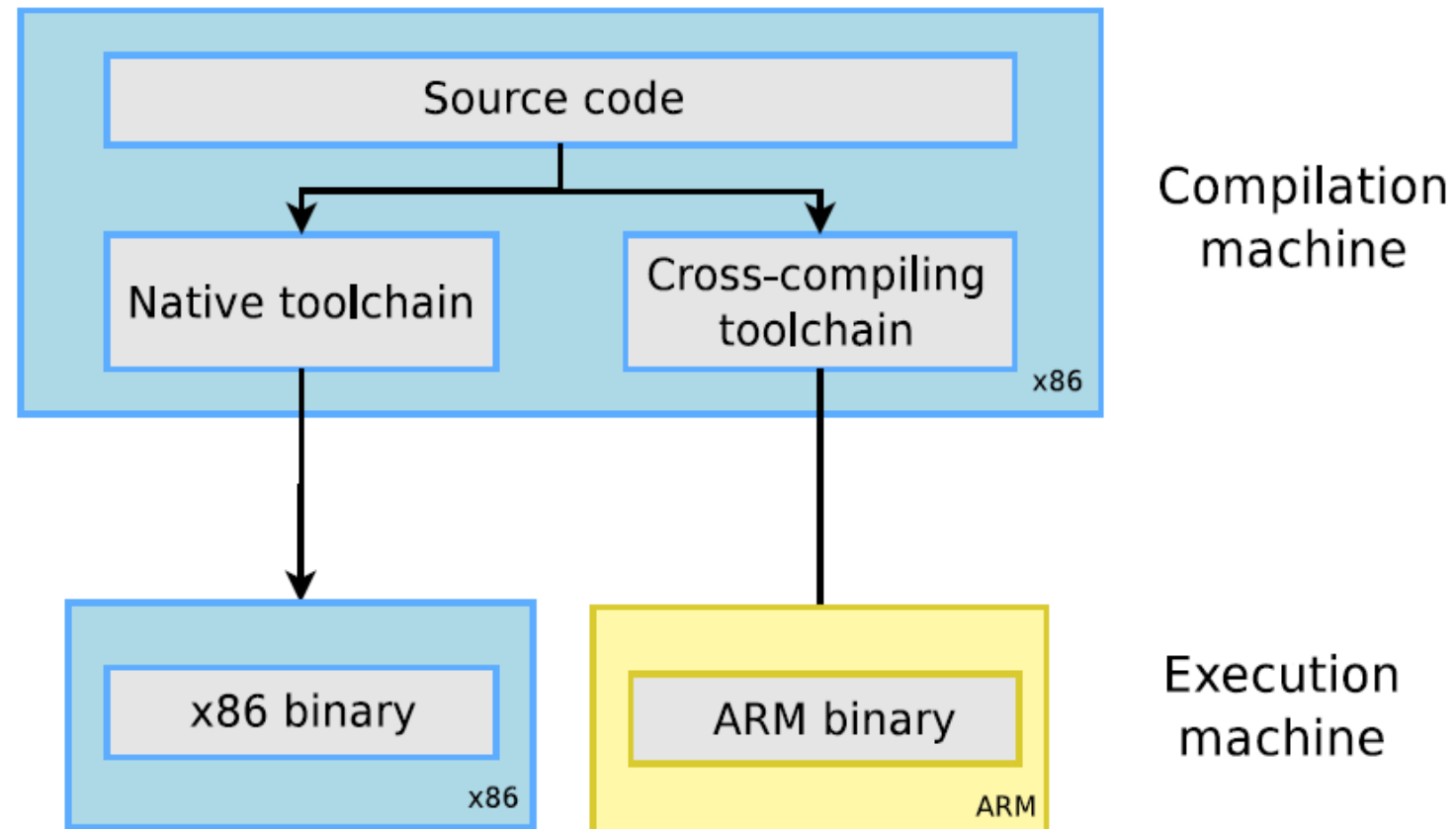
- Though **C** and **embedded C** appear different and are used in different contexts, they have more similarities than the differences. Most of the constructs are same, the difference lies in their applications.
- **C** is used for desktop computers, while **Embedded C** is for micro-controller based applications.
- **C** programmer has the luxury to use resources of a desktop PC like memory, OS, etc. While programming on desktop systems, we need not bother about memory.
- **Embedded** programmer has to use with the limited resources (RAM, ROM, I/O's) on an embedded platforms Thus, program code must fit into the available program memory. If code exceeds the limit, the system is likely to crash.

- Embedded systems often have the real-time constraints, which is usually not there with desktop computer applications.
- So, what basically is different while programming with **Embedded C** is the mind-set for embedded applications, we need to optimally use the resources, make the program code efficient, and satisfy real time constraints, if any. All this is done using the basic constructs, syntaxes, and function libraries of 'C'.

- Regular C programming needs **native compiler** runs on your workstation or your PC and generates code for your workstation, usually x86 like Intel and AMD.
- Embedded C programming needs **Cross Compiler** that runs on the development machine, but generates code for the target.

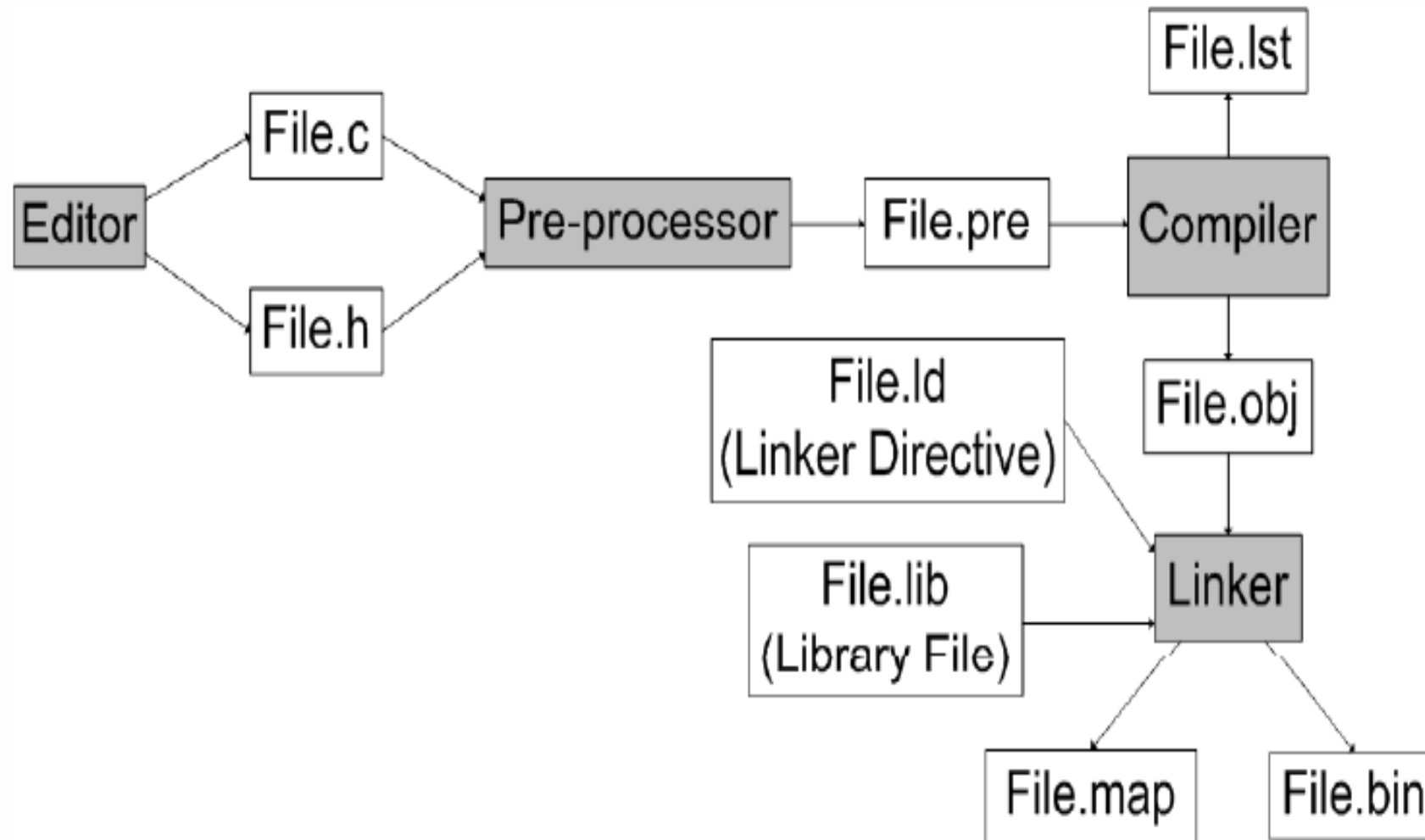


- Development Environment is different from target environment.
- Need for cross platform development and debugging tools.



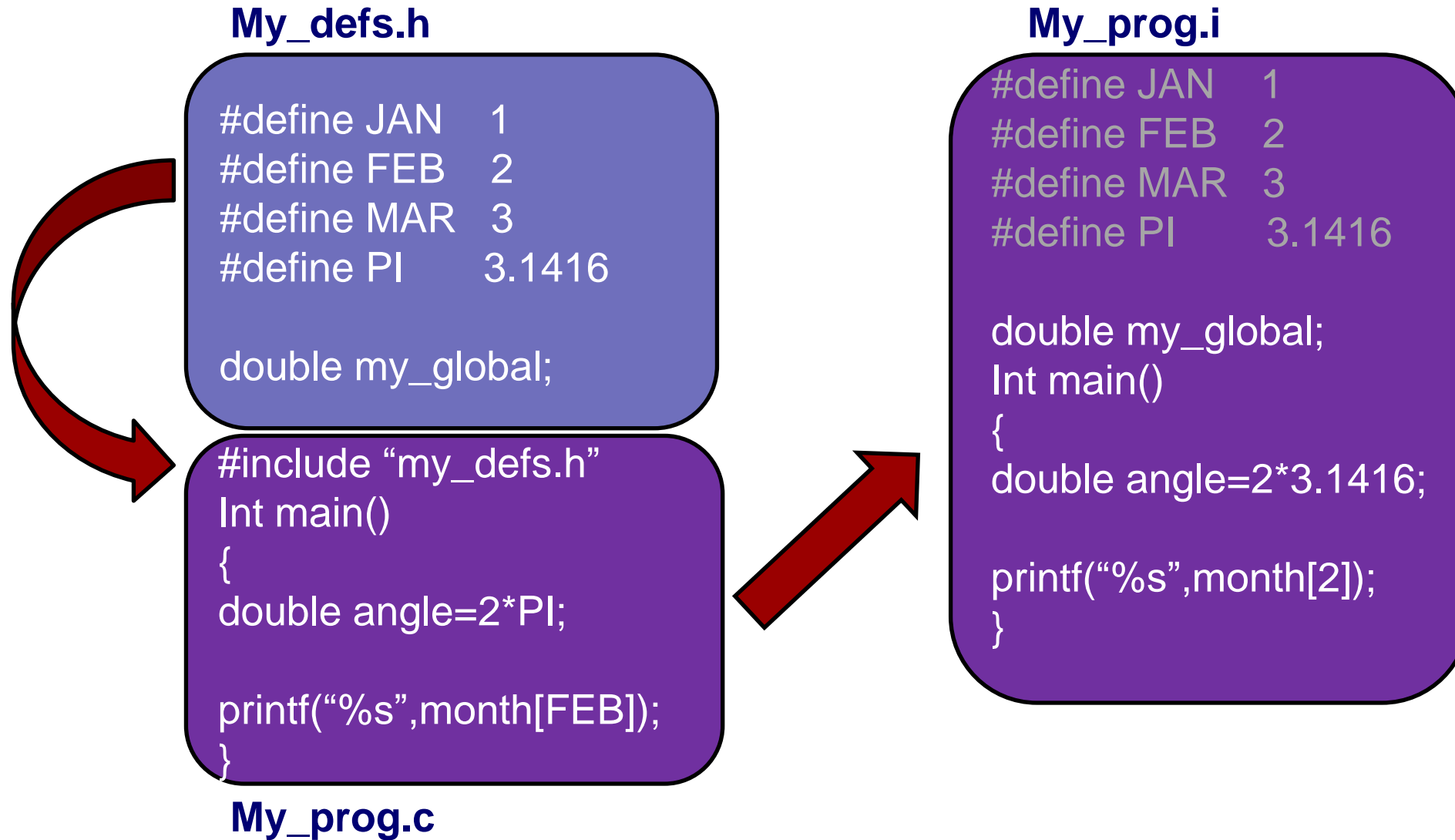
▪ **The aim of Embedded C is**

- Reliability
- Portability
- Maintainability
- Testability
- Reusability
- Extensibility
- Readability
- Optimization.



▪ Preprocessor

- The input to this phase is the .c File and .h Files
- The preprocess process the preprocessor keywords like #define, #ifdef, #include, etc. and generate a new **.pre file** or **.i file** after the text replacement process.
- The output of this phase is a C Code without any preprocessor keyword.



■ Compiler

- The input to this phase C Files without any '#' statement.
- The compiler parse the code, and check the syntax correctness of the file.
- Convert the C Code into optimized machine language code.
- The output of this phase is object file **.o file** or **.obj file** and **list file (.lst file or .lss file)**.
- List File: Contains the corresponding assembly code for each line.

▪ **Assembler**

- The input to this phase .asm File
- The Assembler converts the assembly code to the corresponding machine language code.
- The output of this phase is object file **.o file** or **.obj file**.

▪ Linker File

- The input file to the linker and it contains the memory mapping information of the segments in the microcontroller memory.
- So using the linker file you could place any segment in any place in the memory.
- `#pragma` is used to change the default segment for specific variable or code

▪ **Linker**

- The input to this phase multiple **.obj Files**.
- The Linker merges different object files and library files.
- Linker is used to be sure that every .obj files or .o files have all the external data and functions that is defined. And allocate target memory (RAM,ROM and Stack) to give addresses to the variables and function according to **.ld file(linker file)** and generate **.map file** and **.bin File**.
- The output of this phase is the **.bin file** and the **.map file**.

- The **bin** File which burn in target . It may has another names (.hex , .out , .elf , .srec "Motorola").
- The Map file format is mainly dependent on the linker, but in general it contains the allocation of different object files in the different memory segments.

- Pre-processor commands start with ‘#’ symbol which may optionally be surrounded by spaces and tabs .
- The pre-processor allows us to:
 - define, test and compare constants.
 - include files.
 - Debug.
 - write macros.
- Note that preprocessor statements are **NOT** terminated by a semicolon. Traditionally.
- Preprocessor statements are listed at the beginning of the source file.

- Preprocessor statements are handled by the preprocessor before the program is actually compiled.
- All ‘#’ statements are processed first. Once these **text replacement** has taken place by the preprocessor, the program is then compiled.

- The **#define** directive is a better way to write the literal constants, as it increase the code readability and maintainability.
 - #define Pi 3.1416
 - #define Num_Of_Emp 10
 - #define Emp_Salary 100
 - #define Total_Salary Num_Of_Emp* Emp_Salary
- These definitions are simply handled as a **text replacement** operation done by the preprocessor before the compilation of the file.
- In the program itself, values cannot be assigned to symbolic constants. It is not a variable.

- Which method **typedef** or **#define** is preferred in declaring data types and why?

```
#define p_NewType struct s_NewType *  
P_NewType p1, p2;
```

```
typedef struct newType * newtype_ptr;  
newtype_ptr p3,p4;
```

- Which method **typedef** or **#define** is preferred in declaring data types and why?

```
#define p_NewType struct s_NewType *  
P_NewType p1, p2;
```

```
typedef struct newType * newtype_ptr;  
newtype_ptr p3,p4;
```

- For **#define** it will be **struct s_NewType * p1, p2;** which defines **p1** to be a pointer to the structure and **p2** to be an actual object of structure, which is probably not what you wanted. The second example (**typedef**) correctly defines **p3** and **p4** to be pointers to structure.

- **#include** directive includes a file into code.
- It has two possible forms:
 #include <file> or #include "file"
 - <file> tells the compiler to look where system include files are held (standard libraries).
 - "file" looks for a file in the current directory where program was run from (user defined libraries)
- Included files usually contain C prototypes and declarations from header files and not (algorithmic) C code.
- The **#include** directive causes the pre-processor to edit in the entire contents of another file.

My_defs.h

```
#define JAN 1
#define FEB 2
#define MAR 3
#define PI 3.1416

double my_global;
```

My_prog.c

```
#include "my_defs.h"
Int main()
{
    double angle=2*PI;

    printf("%s",month[FEB]);
}
```

My_prog.i

```
#define JAN 1
#define FEB 2
#define MAR 3
#define PI 3.1416

double my_global;
Int main()
{
    double angle=2*3.1416;

    printf("%s",month[2]);
}
```

- The preprocessor supports a macro facility which should be used with care.
- Parameterized macro is a type of macro that is able to insert given objects into its expansion. This gives the macro some of the power of a function.

```
#define MAX(A,B) A > B ? A : B
#define MIN(X,Y) ((X) < (Y) ? (X) : (Y))
Int main(){
    int i = 10, j = 12, k;
    k = MAX(i, j);      printf("k = %d\n", k);
    k = MAX(j, i) * 2;   printf("k = %d\n", k);
    k = MIN(i, j) * 3;   printf("k = %d\n", k);
    k = MIN(i--, j++);   printf("i = %d\n", i);
}
```

- The preprocessor supports a macro facility which should be used with care.
- Parameterized macro is a type of macro that is able to insert given objects into its expansion. This gives the macro some of the power of a function.

```
#define MAX(A,B) A > B ? A : B
#define MIN(X,Y) ((X) < (Y) ? (X) : (Y))
Int main(){
    int i = 10, j = 12, k;
    k = MAX(i, j);      printf("k = %d\n", k);
    k = MAX(j, i) * 2;   printf("k = %d\n", k);
    k = MIN(i, j) * 3;   printf("k = %d\n", k);
    k = MIN(i--, j++);   printf("i = %d\n", i);
}
```

Results

```
k = 12
k = 12
k = 30
i = 8
```

1. Macros is just a text replacement so there is no overhead like functions due to function calling. No need to save the current context when calling a Macro.
2. Macros increase code size.
3. No type checking for the passing parameters like functions.
4. Return not exist like functions.
5. Compiler errors within macros are often difficult to understand, because they refer to the expanded code, rather than the code the programmer typed. Unlike functions error appears at the same line which the error exist.

6. Use **Macros** to replace tiny functions or a bulk of code that are called or used so many times like inside a loop to save calling time and improve performance.
7. **Always** surround all symbols inside your macro with parenthesis to avoid any future mistakes due to Using this macro.
 - Example:

```
#define MUL(x,y) ((x)*(y))
```

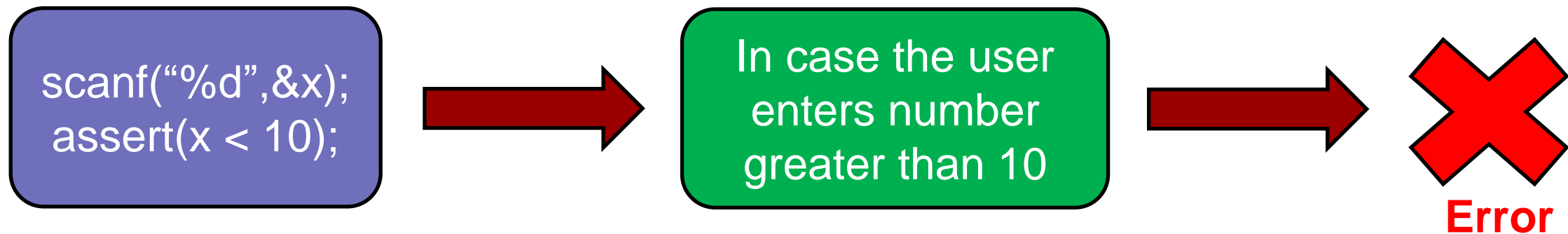
- Inline function is a function upon which the programmer has requested that the compiler insert the complete body of the function in every place that the function is called, rather than generating code to call the function in the one place it is defined(text replacement function). (However, compilers are not obligated to respect this request.) due to some constraints according to each compiler.
- Example:

```
inline int max(int a, int b)
{
    return (a > b) ? a : b;
}
```

- Use of inline functions provides several benefits over macros:
 1. Inline function is just a text replacement like Macro so there is no overhead like functions due to function calling. No need to save the current context when calling a Macro.
 2. Macro invocations do not perform type checking, or even check that arguments are well-formed, whereas function and inline function calls usually do.
 3. Macros cannot use the return keyword with the same meaning as a function and inline function would do.

4. If any error occurs when using inline function it will appear at the line which the error occurs. So Debugging information for inline code is usually more helpful than that of macro-expanded code

- Header <assert.h>
- Following is the declaration for assert() Macro.
 - `void assert(int expression);`
- **expression** can be a variable or any C expression.
If **expression** evaluates to **TRUE**, assert() does nothing.
If **expression** evaluates to **FALSE**, assert() displays an error message on stderr and aborts program execution.
- This macro does not return any value.



- It is a preprocessor keyword.
- Normally it is used to add or delete some parts of code based on a certain condition.
- The #if only deals with definition as it is evaluated before the compilation of the program

```
#define log 1
int main(){
    int x=1;
    printf("%d\n",x);
    #if (log==1)
        printf("Hello World!\n");
    #else
        printf("Hello Earth!\n");
    #endif
    return 0;
}
```



**“Hello World!”
will be printed**

- It is a preprocessor keyword.
- Normally it is used to add or delete some parts of code based on the **existence** of a certain definition.
- Allows a section of a program to be compiled only if the macro that is specified as the parameter has been defined, no matter which its value is.

```
#define log
int main(){
int x=1;
Printf("%d\n",x);
#ifdef log
    printf("Hello World!\n");
#else
    printf("Hello Earth!\n");
#endif
return 0'
}
```



**“Hello World!”
will be printed**

- One of the advantages of the C programming language is that it is a configurable language.
- Can choose to control the DC Motor speed using PWM or SPI by changing only one line of code.

```
#define PWM_SUPPORTED

Void MotorSpeed() {
    # ifdef PWM_SUPPORTED
        PWM_Change();
    #else
        SPI_Change();
    #endif
}
```

```
#ifdef MACRO_NAME1
    #ifdef MACRO_NAME2
        #ifdef MACRO_NAME3
            printf("MACRO_NAME1, MACRO_NAME2, and MACRO_NAME3\n");
        #else
            printf("MACRO_NAME1 and MACRO_NAME2\n");
        #endif
    #else
        printf("MACRO_NAME1\n");
    #endif
#else
    printf("No macro name defined.\n");
#endif
```

The #ifndef Statement

- It is a preprocessor keyword.
- Normally it is used to add or delete some parts of code based on the **inexistence** of a certain definition.
- the code is only compiled if the specified identifier has not been previously defined.

```
#define log
int main(){
int x=1;
Printf("%d\n",x);
#ifndef log
    printf("Hello World!\n");
#else
    printf("Hello Earth!\n");
#endif
return 0'
}
```



**“Hello Earth!”
will be printed**

- The **#error** directive outputs a message through the error stream.
- This directive aborts the compilation process when it is found, generating a compilation error message that can be specified as its parameter.
- Example:

#error "Code is not completed yet"

- This directive is used to specify diverse options to the compiler. These options are specific for the platform and the compiler you use.
- Consult the manual or the reference of your compiler for more information on the possible parameters that you can define with #pragma.
- A #pragma statement can inform the compiler that the target chip does have a certain optional instruction feature, and that it can therefore optimize code that will benefit from the instruction. These examples are present in the header file of the MC68HC05C8 freescale μ C.

```
#pragma has MUL;  
#pragma has WAIT;  
#pragma has STOP;
```


- #pragma port directives describe the ports available on the target computer. This declaration reserves memory-mapped port locations, so the compiler does not use them for data memory allocation.

```
#pragma portw PORTA @ 0x0000  
#pragma portw PORTB @ 0x0001;  
#pragma portw DDRA @ 0x0004;  
#pragma portw DDRB @ 0x0005;
```

- Most C compilers for the M68HC05 family provide automatic placement of variables in the available RAM of the part. Specific memory addresses are identified to the compiler by the #pragma memory directives. The following code segment shows an example of how the memory is defined within an M68HC05 program:

```
#pragma memory ROMPAGE0 [48] @ 32;  
#pragma memory ROMPROG [5888] @ 2048;  
#pragma memory RAMPAGE0 [176] @ 80;  
#pragma memory RAMPROG [256] @ 256; //EEPROM
```

- It is compiler depended
- The **#asm** and the **#endasm** are Most common. The code enclosed in a block that starts with **#asm** and ends with **#endasm** must be in standard assembly language.
- The compiler will not attempt to optimize such code. The compiler assumes that the user has a good reason to avoid the compiler's code generation and optimization.
- Variables defined in the C program can be used safely in the assembly code with the same name.

- Example for a delay function for Motorola freescale 68HC705C8 assembly language.

```
void wait(int delay)
{
    #asm
    LOOP:
        DEC delay;
        BNE LOOP;
    #endasm
}
```

- Example for function gets the running core id for Quad-core arm cortex-a9 processor.
- The core id stored in the first two bits of MPIDR register.

```
unsigned int get_cpu_id(void)
{
    unsigned int id = 0;
    // Read MPIDR Register value
    asm volatile("MRC p15,0,%0,c0,c0,5;" : "=r" (id));
    // Extract Bits 0 & 1
    id = (id & 3);
    return id;
}
```

- Naming Convention is to put a standard way for naming your program Variables, Functions and even comments.
- Identifier name should provide additional information about the its usage and its functionality. It will provide better understanding in case of code reuse after a long interval of time.
- Reasons for using a naming convention:
 - To reduce the effort needed to read and understand source code.
 - To enhance source code appearance.

- The following is an example of naming convention rules:
 - Underscores to split words in structures or function names as most programming languages do not allow whitespace in identifier names.
 - Functions, structures, unions and enums names are starting with lower-case letter.
 - members names of unions, structures and enums typically are in lower case.
 - Put prefix for pointers, global data and structure data.
Example: **int *p_pointer_name;**

much easier to
read because it is
indented.

```
if (hours < 24 && minutes < 60)
{
    return true;
}
else
{
    return false;
}
```

```
if ( hours < 24
    && minutes < 60
)
{return true
;} else
{return false
;}
```


- Using meaningful comments is very useful.
- It helps you to fix your code faster.
- It helps you to modify someone else code.
- It helps others understand your code and learn from it.
- Provide comments at the beginning of each file to describe the purpose of this file.
- Provide comments at the beginning of each function to describe the purpose of this file, its input arguments, its output and any other additional notes.
- Provide comments all over the code to make it understandable.

Function Comments Example

```
/******  
* Purpose : This subroutine computes the area of a square room whose side is given.  
* Input : the side of the square  
* Output : the area of the square whose side is less than 25  
* Notes : The side has to be positive and less than or equal to 25 or error occurs.  
*****/  
unsigned long calculate_square_area(unsigned long square_side)  
{  
    unsigned long area_result;  
    if (square_side <= 25)  
    {  
        area_result = square_side * square_side; // compute valid area result  
    }  
    else  
    {  
        area_result = 0; // indicate invalid area result  
        g_error = g_error + 1; // increment global error count  
    }  
    return(area_result);  
}
```

```
/******  
* Filename: Module.c  
* Author: you  
* Date: 1/1/2014  
* Version: 2.1  
* Description: main application  
*****/  
  
/*includes*/  
/*Global variables*/  
/*ISR's*/  
Void main()  
{  
    /*Initialization Code*/  
    While(1)  
    {  
        /*Call Application*/  
    }  
}
```

```
For(;;){  
  
    /* code */  
  
}
```

Better performance

```
While(1){  
  
    /* code */  
  
}
```

Most common

```
Loop:  
  
    /* code */  
  
    goto loop;
```

Don't use it

C Data Types

Type	Bits	Bytes	Range
char	8	1	-128 → +127
Unsigned char	8	1	0 → +255
short	16	2	-32768 → +32767
Unsigned short	16	2	0 → +65535
Int	16	2	-32768 → +32767
Unsigned int	16	2	0 → +65535
Long	32	4	-2147483648 → +2147483647
Unsigned long	32	4	0 → +4294967295
Long long	64	8	
enum	16	2	-32768 → +32767

- a. `int a;` // An integer
- b. `Const int a;` // constant int
- c. `int *a;` // A pointer to an integer
- d. `int **a;` // A pointer to a pointer to an integer
- e. `int a[10];` // An array of 10 integers
- f. `int *a[10];` // An array of 10 pointers to integers
- g. `int (*a)[10];` // A pointer to an array of 10 integers
- h. `int (*a)(int);` // A pointer to a function a that takes an integer argument and returns an integer
- i. `int (*a[10])(int);` // An array of 10 pointers to functions that take an integer argument and return an integer

- **Important Notes:**

1. **No float:**

Because the HW must contains FPU to support floating point operations.

2. **No Integer:**

Because there is no rule for integer size some compilers define integer numbers as 16-bits(2bytes) and other define them as 32-bits(4 bytes).

- Declaration of a variable in C hints the compiler about the type and size of the variable in compile time. Similarly, declaration of a function(prototype) hints about return type and number of function parameters.
- No space is reserved in memory for any variable in case of declaration.
- **Example 1:** `int a;`
Here variable 'a' is declared of data type “int”
- **Example 2:** `int add(int a,int b);`
Here a function prototype “declaration”.

- Defining a variable means declaring it and also allocating space to hold it. Similarly define a function means write its body code. We can say "Definition = Declaration + Space reservation".

- **Example 1:** `int a = 10;`

Here variable "a" is described as an int to the compiler and memory is allocated to hold value 10.

- **Example 2:** Here is a function code body “definition”

```
int add(int a,int b)
{
    return (a+b);
}
```

- Apply to any “integral” data type like char, short, int and long.
- Arguments are treated as bit vectors.
- Operations applied bitwise.

Operator	Symbol
AND	&
OR	
XOR	^
One's complement	~
Shift left	<<
Shift right	>>

$C = A \ \& \ B;$
(AND)

A	0	1	1	0	0	1	1	0
B	1	0	1	1	0	0	1	1
C	0	0	1	0	0	0	1	0

$C = A \ | \ B;$
(OR)

A	0	1	1	0	0	1	0	0
B	0	0	0	1	0	0	0	0
C	0	1	1	1	0	1	0	0

$C = A \ ^ \wedge \ B;$
(XOR)

A	0	1	1	0	0	1	0	0
B	1	0	1	1	0	0	1	1
C	1	1	0	1	0	1	1	1

$B = \sim A;$
(COMPLEMENT)

A	0	1	1	0	0	1	0	0
B	1	0	0	1	1	0	1	1

$B = A \ll 3;$

(Left shift 3 bits)

A	1	0	1	0	1	1	0	1
B	0	1	1	0	1	0	0	0

$B = A \gg 2;$

(Right shift 2 bits)

A	1	0	1	1	0	1	0	1
B	0	0	1	0	1	1	0	1

$B = '1';$

$C = '5';$

$D = (B \ll 4) \mid (C \& 0x0F);$

($B \ll 4$)

($C \& 0x0F$)

D

$B = 0\ 0\ 1\ 1\ 0\ 0\ 0\ 1$ (ASCII 0x31)

$C = 0\ 0\ 1\ 1\ 0\ 1\ 0\ 1$ (ASCII 0x35)

$= 0\ 0\ 0\ 1\ 0\ 0\ 0\ 0$

$= 0\ 0\ 0\ 0\ 0\ 1\ 0\ 1$

$= 0\ 0\ 0\ 1\ 0\ 1\ 0\ 1$ (Packed BCD 0x15)

- **Don't confuse** between & (bit wise operator) and && (logical operator)
1 (01) & 2(10) \rightarrow 0 *whereas* 1 && 2 \rightarrow true.
- **Note:**

$$x \gg y = x / 2^y \quad \text{and} \quad x \ll y = x * 2^y$$

- **Set specified bit in Register**

Make OR operation on the register with The pin number.

```
#define SET_BIT(REG,BIT_NUM) (REG = REG |  
(1 << BIT_NUM))
```

- **Clear specified bit in Register**

Make AND operation on the register with (NOT) The pin number.

```
#define CLEAR_BIT(REG,BIT_NUM) (REG = REG & ~(1 <<  
BIT_NUM)))
```

- **Toggle specified bit in Register**

Make XOR operation on the register with The bit number.

```
#define TOGGLE_BIT(REG,BIT_NUM) (REG = REG ^  
(1 << BIT_NUM))
```

- **Rotate bits right in Register**

Make shift right operation on the register with the required rotate right number and OR the result with the result of shifting the register left with register size – required number of rotate.

```
#define ROR(REG,BIT_NUM) (REG =  
(REG >> BIT_NUM) | (REG << (8-BIT_NUM)))
```

- **Rotate bits Left in Register**

Make shift left operation on the register with the required rotate left number and OR the result with the result of shifting the register right with register size – required number of rotate.

```
#define ROL(REG,BIT_NUM) (REG =  
(REG << BIT_NUM) | (REG >> (8-BIT_NUM)))
```

- **Check if a specific bit is set in Register**

```
#define BIT_IS_SET(REG,BIT_NUM) (REG & (1<<BIT_NUM))
```

- **Check if a specific bit is cleared in Register**

```
#define BIT_IS_CLEAR(REG,BIT_NUM) ( (REG & (1<<BIT_NUM))  
== 0)
```


- The region in where the identifier can be referenced and valid.
 1. **Function or Block Scope**

Local variable inside function or block.
 2. **File Scope**

Global variable inside file. Variable can be accessed by all the functions in the program file.
 3. **Software Scope**

Global variable can be manipulated by multiple of files in the same project using “extern” keyword.

- Also called “Storage Duration”.
- Period during which the identifier exists in memory

1. Static and Global allocation

Exist in memory for the entire execution of a program.

2. Automatic or local allocation

Variables are created when a function/block is entered and destroyed when the function/block is exited

3. Dynamic allocation

The memory is allocated in a different memory area called heap.
The lifetime is under the control of the programmer rather than the C run-time system.

- Variables which are declared above the main() function.
- Variables declared outside any procedures.
- These variables are accessible throughout the program. Its **life time** equals to the **whole program life**.
- They can be accessed by all the functions in the program → **file scope**. And it may be accessed by other files using extern keyword → **SW scope**.
- Its default value is zero or garbage depend on the compiler.

```
int my_global;
```

```
Void main(){  
    /* code */  
}
```

```
#include <stdio.h>
int x = 0;
void increment(void)
{
    x = x + 1;
    printf("\n value of x: %d\n", x);
}
int main()
{
    printf("\n value of x: %d\n", x);
    increment();
    return 0;
}
```



Results

value of x: 0
value of x: 1

- An identifier's **storage class** determines its storage duration and scope.
- C provides four storage classes, indicated by the **storage class specifiers**:
 1. auto.
 2. register.
 3. extern.
 4. static.

- Local variables only variables can have automatic storage duration.
- Local variables have automatic storage duration by default, so keyword `auto` is rarely used.
- Local variables are declared at the start of a program's block or function such as in the curly braces `{ }`.
- Function pass parameters are also local variables to the function.
- Local variables are allocated in the stack memory.
- Memory is allocated automatically upon entry to a block and freed automatically upon exit from the block.

- The scope of automatic variables is local to the **block** in which they are declared, including any blocks nested within that block. For these reasons, they are also called **local variables**.
- No block outside the defining block may have direct access to automatic local variables.
- Its default value is garbage.

```
Void main()  
{  
    auto double x,y;  
  
    /* code */  
}
```

- Can only be used for **automatic/Local** variables.
- Register variables are stored in the CPU registers.
- Variable stored in a CPU register can always be accessed faster than the one that is stored in memory. Therefore, if a variable is used at many places in a program, it is better to declare its storage class as register. Its default value is a garbage value.
- Scope of a register variable is local to the block in which it is **defined(local variable)**.
- Lifetime is till control remains within the block in which the register variable is defined.

- The compiler may ignore register declarations. For example, there may not be a sufficient number of registers available for the compiler to use. The following declaration suggests that the integer variable counter be placed in one of the CPU registers and initialized to 1.

```
Void main()
{
    register unsigned char counter = 1;

    /* code */
}
```

- Used when we have to refer to a function or a variable that is implemented in other file in the same project.
- The extern keyword means "**declare without defining**".
- The declaration informs the compiler that a variable by that name and type exists, but the compiler need not allocate memory for it since it is allocated elsewhere.
- The scope of external variable is **SW Scope**. And its life time equals to the **whole program life time**.
- If the global variable not initialized its default value after extern is **Zero**.
- If there are 2 functions or variables In two different C file having the same name. If the programmer try to use "extern" keyword it will produce linker error.

File1.c

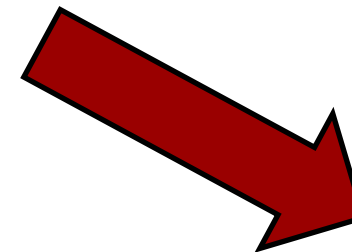
```
#include<stdio.h>

extern int x,y;

Void main()
{
    printf(" x=%d \t y=%d\n",x,y);
}
```

File2.c

```
int x=3;
int y;
```



Results

x=3 y=0

File1.c

```
#include<stdio.h>

extern int f1(void);

void main (void)
{
    int x;
    x = 10*f1();
    printf("x=%d",x);
}
```



File2.c

```
int f1(void)
{
    return 10;
}
```



Results

x=100

- Static is used in the declarations of identifiers for variables and functions.
- Static keyword has two meanings, depending on where the static variable is declared.
- Static Global variables are same as ordinary global variable except that it can not be accessed by other files in the same program even with the use of keyword “extern”.
- Static global variable has the same lifetime like ordinary global variable. But its scope is localized to this file only.

File1.c

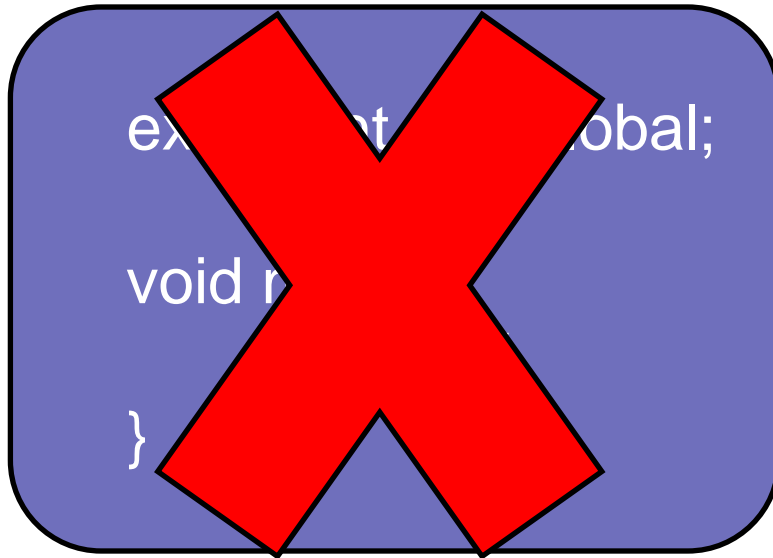
```
extern int my_global;  
  
void main(){  
    /* code */  
}
```



File2.c

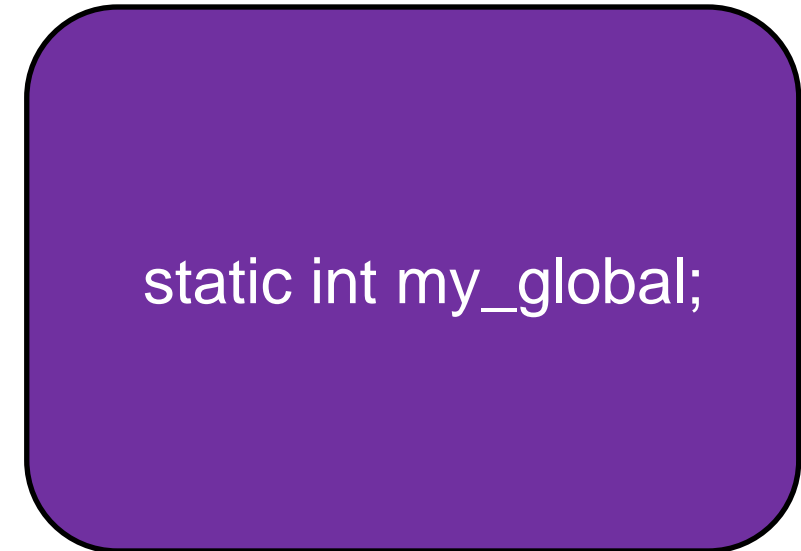
```
static int my_global;
```

File1.c



Linker Error

File2.c



- Static local variables Inside a function:
 1. Static variable is still local to this function.
 2. Static variable is initialized only once no matter how many times that function is used.
 3. Static variable life time equals to the whole program life. And exist in memory for the entire execution of a program.
- Static variable default value is Zero.
- Functions declared static within a file can only be called by other functions within that file. That is, the scope of the function is localized to the module within which it is declared.

Static Local Variable Example

```
#include<stdio.h>
void count_func(vodi)
{
    static int count1 = 0;
    int count2 = 0;
    count1++;
    count2++;
    printf(" %d \t %d\n",count1,count2);
}

Void main()
{
    count_func();
    count_func();
    count_func();
}
```



Results

1	1
2	1
3	1

File1.c

```
#include<stdio.h>

extern int f1(void);
void main (void)
{
    int x;
    x = 10*f1();
    printf("x=%d",x);
}
```



File2.c

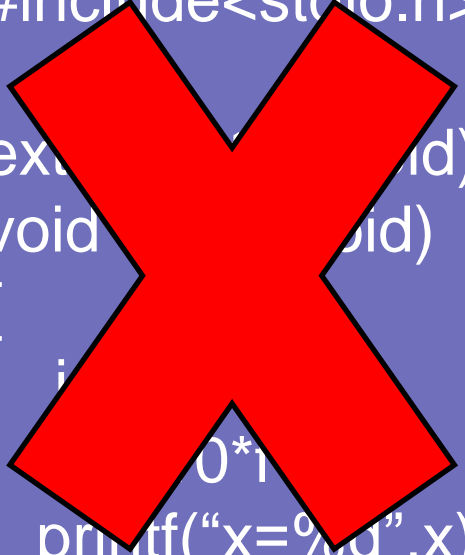
```
static int f1(void)
{
    return 10;
}

int f2(void)
{
    return (f1()*5);
}
```

f2() function
can call f1()
function without
any problem.

File1.c

```
#include<stdio.h>
extern void f1(void);
void f2(void)
{
    int x = 0;
    printf("x=%d",x);
}
```

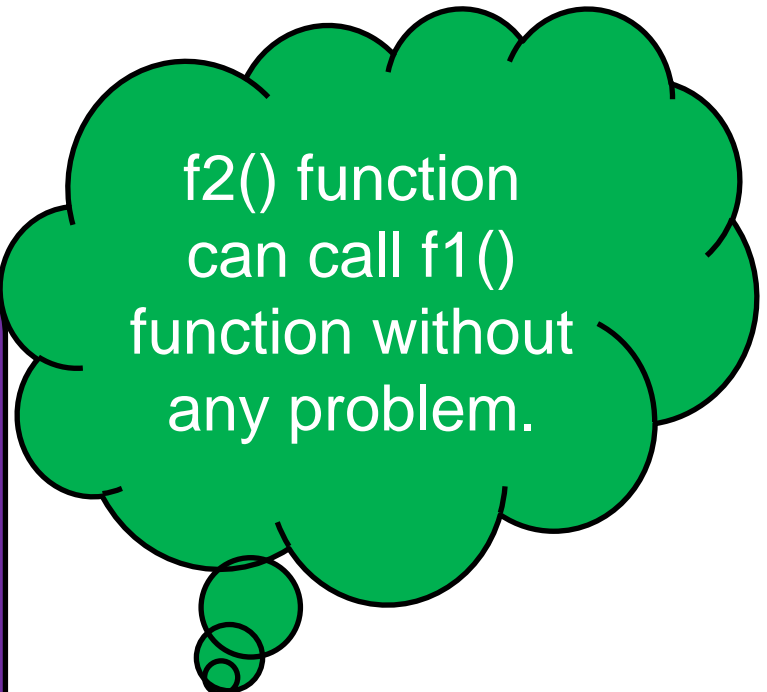


Linker Error

File2.c

```
static int f1(void)
{
    return 10;
}

int f2(void)
{
    return f1()*5;
}
```



f2() function
can call f1()
function without
any problem.

- Volatile keyword is intended to prevent the compiler from applying any optimizations on the code that assume values of variables cannot change on their own.
- Tell the compiler that the value of the variable may change at any time without any action being taken by the code.
- Using with:
 1. Memory mapped HW peripheral registers.
 2. Global variables modified by ISR.
 3. Global Variables shared by multiple tasks in a multi-threaded application.

```
PORTA=0xFF;  
_delay_ms(100);  
PORTA=0x00;
```



Compiler
optimizer will
make it
PORTA=0x00;



To solve it HW
Register must
declare as volatile

```
/* check if a bit 2 is  
set in HW I/O PORT  
Register PORTA or  
the switch is  
pressed */
```

```
If(PINA & (1<<2))  
{  
    /* code */  
}
```



Compiler
Optimizer find
that the if
condition will
never be true
and remove it



To solve it HW
Register must
declare as volatile

```
int flag =0;
ISR(INT0_vect){
    flag++;
}
main(){
    while (flag !=0)
    {
        /* code */
    }
}
```



Compiler
Optimizer find
that while
loop
unreachable
code &
remove it



To solve it Global
variable which
used inside ISR
must be declared
as volatile

```
int cntr=0;

void task1(void) {
    while(cntr != 0)
    {
        /* code */
    }
    /* code */
}

void task2(void) {
    cntr++;
    /* code */
}
```



Compiler
Optimizer find
the while loop
unreachable
code & remove
it



To solve it
Global variable
which changed
by another task
must declare as
volatile

- The **const qualifier** enables you to inform the compiler that the value of a particular variable should not be modified.
- The values of constant variables not changeable through out the program.
- Example:

```
const int x = 10; //must be initialized
```

```
int const x = 10; //the same.
```


▪ A non constant pointer to constant data

- Can be modified to point to any data item of the appropriate type, but the data to which it points cannot be modified by the pointer.
- Example:

```
int x = 10, y = 15;  
const int *ptr;  
ptr = &x;  
ptr = &y;  
y = 20;  
*ptr = 5; //error
```

- Such a pointer might be used to receive an array argument to a function that will process each element without modifying the data.

```
void print_characters( const char *arr_ptr )
{
    while((*arr_ptr) != '\0')
    {
        printf("%c",*arr_ptr);
        arr_ptr++;
    }
}
```

▪ **Passing Structure by address without allowing data modification:**

- Structures are always passed by value a copy of the entire structure is passed. This requires the execution-time overhead of making a copy of each data item in the structure and storing it on the computer's function call stack. When structure data must be passed to a function, we can use pointers to constant data to get the performance of call-by-reference and the protection of call-by-value. When a pointer to a structure is passed, only a copy of the address at which the structure is stored must be made. On a machine with 4-byte addresses, a copy of 4 bytes of memory is made rather than a copy of possibly hundreds or thousands of bytes of the structure.

▪ A constant pointer to non constant data

- A constant pointer to non-constant data always points to the same memory location, and the data at that location can be modified through the pointer.
- This is the default for an array name. An array name is a constant pointer to the beginning of the array. All data in the array can be accessed and changed by using the array name and array subscripting.
- Pointers that are declared const **must be initialized when they're defined.**

- **Example**

```
int x = 10,y = 15;  
int * const ptr = &x;  
*ptr = 5; // x=5  
ptr = &y; //error
```

```
void print_characters(char * const arr_ptr )  
{  
    int i;  
    while(arr_ptr[i] != '\0')  
    {  
        printf("%c",arr_ptr[i]);  
        i++;  
    }  
}
```

▪ A constant pointer to constant data

- The least access privilege is granted by a constant pointer to constant data.
- Such a pointer always points to the same memory location, and the data at that memory location cannot be modified.
- Example

```
int x = 10,y = 15;  
const int * const ptr = &x;  
x= 3;  
*ptr = 5; // error  
ptr = &y; //error
```

- **Can a parameter be both const and volatile ?**

Yes. An example is a read-only status register like **SREG** register in AVR Microcontrollers . It is volatile because it can change unexpectedly. It is const because the program should not attempt to modify it.

- **Can a pointer be volatile ?**

Yes, although this is not very common. An example is when an interrupt service routine modifies a pointer to a buffer.

- Collections of related variables under one name. Structures may contain variables of many different data types in contrast to arrays that contain only elements of the same data type.
- Structure is not a variable declaration, but a type declaration
- The variables in a structure are called **members** and may have any data type, including int or char or arrays or other structures.
- Each structure definition must end with a semicolon.

- Example of employee structure data type declaration:

```
struct employee {  
    char firstName[ 20 ];  
    char lastName[ 20 ];  
    int age;  
    char gender;  
    double hourly_salary;  
};
```

- Structure variables(objects or instants) are declared like variables of other types:

```
struct employee emp1,emp2;
```

- In C Programming for both the structure type declaration and the structure variable declaration the keyword **struct** must precede the meaningful name you chose for the structure type.
- You can declare a structure type and variables simultaneously.

```
struct employee {  
    char firstName[ 20 ];  
    char lastName[ 20 ];  
    int age;  
    char gender;  
    double hourly_salary;  
}emp1,emp2;
```

- To access a given member the **dot notation** is use. The “dot” is officially called the **member access operator**.

`emp1.age = 25;`

- structure member can be treated the same as any other variable of that type. For example the following code:

`emp2.age = emp1.age + 2;`

- Structure variables can also be assigned to each other, just like with other variable types:

`emp2 = emp1;`

- each member of **emp2** gets assigned the value of the corresponding member of **emp1**.

- Structure members can be **initialized at declaration**. This is similar to the initialization of arrays. the initial values are simply listed inside a pair of braces, with each value separated by a comma.

```
struct employee person = {"Ahmed", "Rafik", 25, 'm', 100.5};
```

- Members of the same structure type must have unique names, but two different structure types may contain members of the same name without conflict.

```
struct fruit
{
    char *name;
    char *color;
}fruit1;
fruit1.name = "apple";
```

```
struct vegetable
{
    char *name;
    char *color;
}veg1;
Veg1.name = "tomatoes"
```

```
#include<stdio.h>
#include<string.h> //for strcpy function
struct student {
    int id;
    char name[30];
};
void main() {
    struct student record[2];
    // 1st student's record
    record[0].id=1;
    strcpy(record[0].name, "Khaled");
    // 2nd student's record
    record[1].id=2;
    strcpy(record[1].name, "Ayman");
    printf("%d %s \n %d %s", record[0].id, record[0].name, record[1].id, record[1].name);
}
```

- One can have pointer variable that contain the address of a structure variable, just like with the basic data types. Structure pointers are declared and used in the same manner as simple pointers:

```
struct employee emp3,*emp_ptr;  
emp_ptr = &emp3;  
(*emp_ptr).age = 22;  
(*emp_ptr).hourly_salary = 70;
```

- The above code has **indirectly initialized** elements of the structure **emp3** through the use of the pointer **emp_ptr**.

- The type of the variable **emp_ptr** is pointer to a **employee** structure.
- In C, there is a special operator ‘->’ which is used as a **shorthand** when working with pointers to structures. It is officially called the **structure pointer operator** or **arrow**.
- the last two lines of the previous example could also have been written as:

```
struct employee emp3,*emp_ptr;  
emp_ptr = &emp3;  
emp_ptr -> age = 22;  
emp_ptr -> hourly_salary = 70;
```

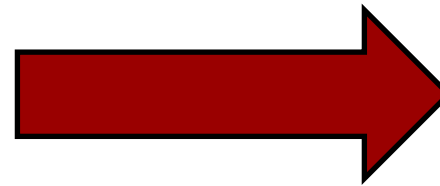
- A structure cannot contain an instance of itself. For example, a variable of type struct employee cannot be declared in the definition for struct employee. A pointer to struct employee, however, may be included.
- **Example:**

```
struct employee {  
    char firstName[ 20 ];  
    char lastName[ 20 ];  
    int age;  
    char gender;  
    double hourlySalary;  
    struct employee emp; //error  
    struct employee emp_ptr; //pointer  
};
```


- Unions are C variables whose syntax look similar to structures, but act in a completely different manner.
- Once a union variable has been declared, the amount of memory reserved is just enough to be able to represent the **largest member**. (Unlike a structure where memory is reserved for **all members**).
- Data actually stored in a union's memory can be the data associated with **any** of its members. But **only one** member of a union can contain valid data at a given point in the program.
- It is the **user's responsibility** to keep track of which type of data has most recently been stored in the union variable.

Unions Example

```
#include <stdio.h>
#include <string.h>
union Data
{
    int i;
    float f;
    char str[20];
};
void main( ){
    union Data d1;
    d1.i = 10;
    printf( "d1.i : %d\n", d1.i);
    d1.f = 220.5;
    printf( "d1.f : %f\n", d1.f);
    strcpy(d1.str,"Embedded C);
    printf( "d1.str : %s\n", d1.str);
    printf("Union size : %d bytes",sizeof(d1));
}
```



Results


```
data.i : 10
data.f : 220.5
data.str : Embedded C
Union size : 20 bytes
```

Unions Example

```
#include <stdio.h>
#include <string.h>
union Data
{
    int i;
    float f;
    char str[20];
};
void main( ){
    union Data d1;
    d1.i = 10;
    d1.f = 220.5;
    strcpy(d1.str,"Embedded C);
    printf( "d1.i : %d\n", d1.i);
    printf( "d1.f : %f\n", d1.f);
    printf( "d1.str : %s\n", d1.str);
    printf("Union size : %d bytes",sizeof(d1));
}
```

Here, we can see that values of i and f members of union got corrupted because final value assigned to the variable has occupied the memory location and this is the reason that the value of str member is getting printed very well.

Results



```
data.i : 1700949317
data.f :
668294057629101740
00000.000000
data.str : Embedded C
Union size : 20 bytes
```

- C language enables you to specify the number of bits in which an **unsigned** or **int** member of a structure or union is stored.
- Bit fields enable better memory utilization by storing data in the minimum number of bits required.
- Bit field does not enable us to access individual bits.
- To define bit fields, Follow **unsigned** or **int** member with a colon (:) and an integer constant representing the width of the field.

```
struct my_bitfield
{
    unsigned short_element: 1; // 1-bit in size
    unsigned long_element: 8; // 8-bits in
    size
};
```

- Unnamed bit field used as padding in the structure and nothing may be stored in these bits.
- unnamed bit field with a zero width is used to align the next bit field on a new storage-unit boundary.

```
struct Example
```

```
{
```

```
    unsigned a : 13;
```

```
    unsigned   : 19;
```

```
    unsigned b :  3;
```

```
};
```

```
struct Example
```

```
{
```

```
    unsigned a : 13;
```

```
    unsigned   :  0;
```

```
    unsigned b :  3;
```

```
};
```

- Used to define the peripheral register bits inside Micro-controller.

```
typedef unsigned char byte;
typedef union {
    byte Data;
    struct {
        byte BIT0 :1; /* Register Bit 0 */
        byte BIT1 :1; /* Register Bit 1 */
        byte BIT2 :1; /* Register Bit 2 */
        byte BIT3 :1; /* Register Bit 3 */
        byte BIT4 :1; /* Register Bit 4 */
        byte BIT5 :1; /* Register Bit 5 */
        byte BIT6 :1; /* Register Bit 6 */
        byte BIT7 :1; /* Register Bit 7 */
    } Bits;
} HW_Register;
```

- An enumeration, introduced by the keyword `enum`, is a set of **integer enumeration constants** represented by identifiers.
- Values in an `enum` start with 0, unless specified otherwise, and are incremented by 1.

```
#define sun 0
#define mon 1
#define tue 2
#define wed 3
#define thu 4
#define fri 5
#define sat 6
void main(){
    int today = sun;
    if(today == mon){
        /* code */
    }
}
```



```
enum day {
    sun, mon, tue, wed, thu, fri, sat
};
void main(){
    enum day today = sun;
    if(today == mon) ){
        /* code */
    }
}
```

- Since the first value in the preceding enumeration is explicitly set to 1, the remaining values are incremented automatically. resulting in the values 1 through 7.

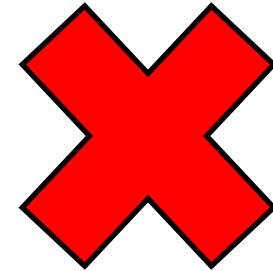
```
enum day {  
sun = 1, mon, tue, wed, thu, fri, sat  
};
```

- The constants used may be specified.

```
enum pc_state {  
shut_down = 1, sleep = 3, hibernate = 5, running = 7  
};
```


- The identifiers in an enumeration must be unique.

```
enum pc_state {  
    shut_down = 1, sleep = 3, sleep = 5, running=7  
};
```



Error

- Multiple members of an enumeration can have the same constant value.

```
enum pc_state {  
    shut_down = 1, sleep = 3, hibernate = 3, running = 7  
};
```

- It is used to define your own data types.
- Example:

```
typedef unsigned char uint8;  
typedef signed char sint8;  
typedef unsigned short uint16;  
typedef signed short sint16;  
typedef unsigned long uint32;  
typedef signed long sint32;  
typedef int* ptoi;
```

```
struct employee{  
    char name[20];  
    int age;  
};  
struct employee e1,*e2;
```

```
struct employee {  
    char name[20];  
    int age;  
};  
typedef struct employee EMP;  
EMP e1,*e2;
```

```
typedef struct{  
    char name[20];  
    int age;  
}EMP;  
EMP e1,*e2;
```

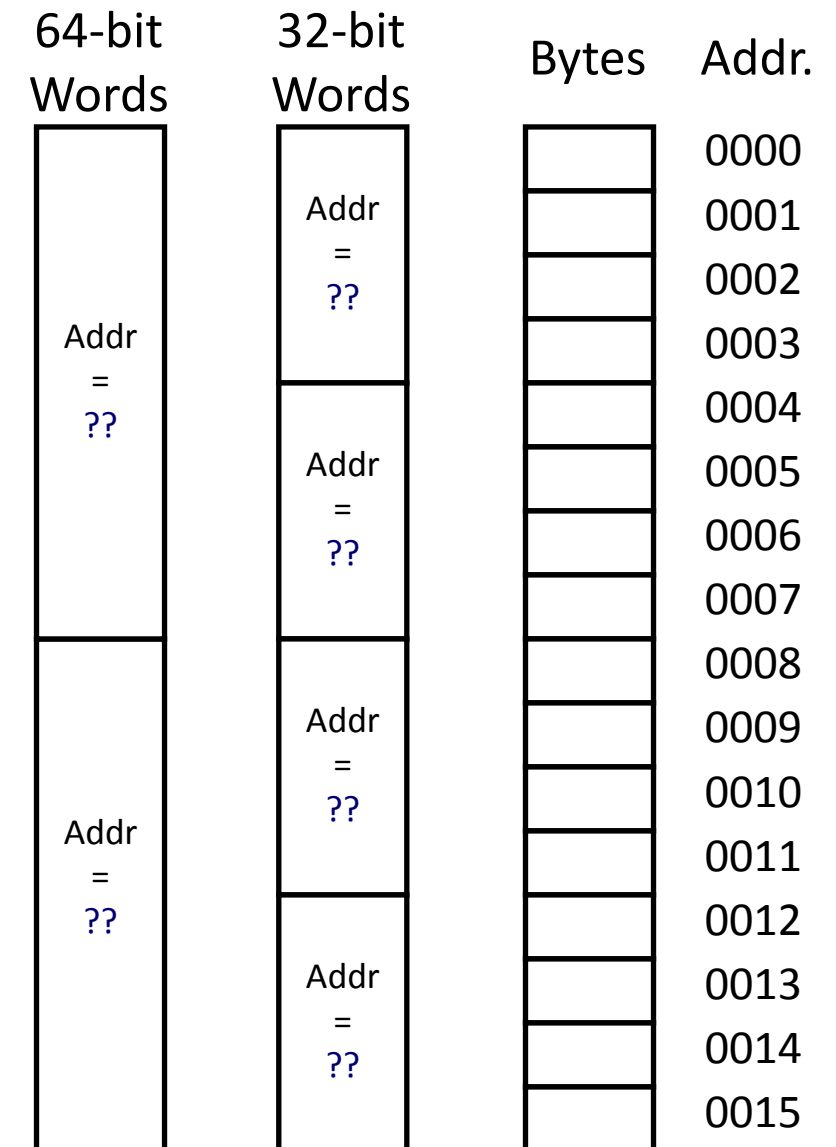
Boolean in C Language

```
#define False 0  
#define True 1
```

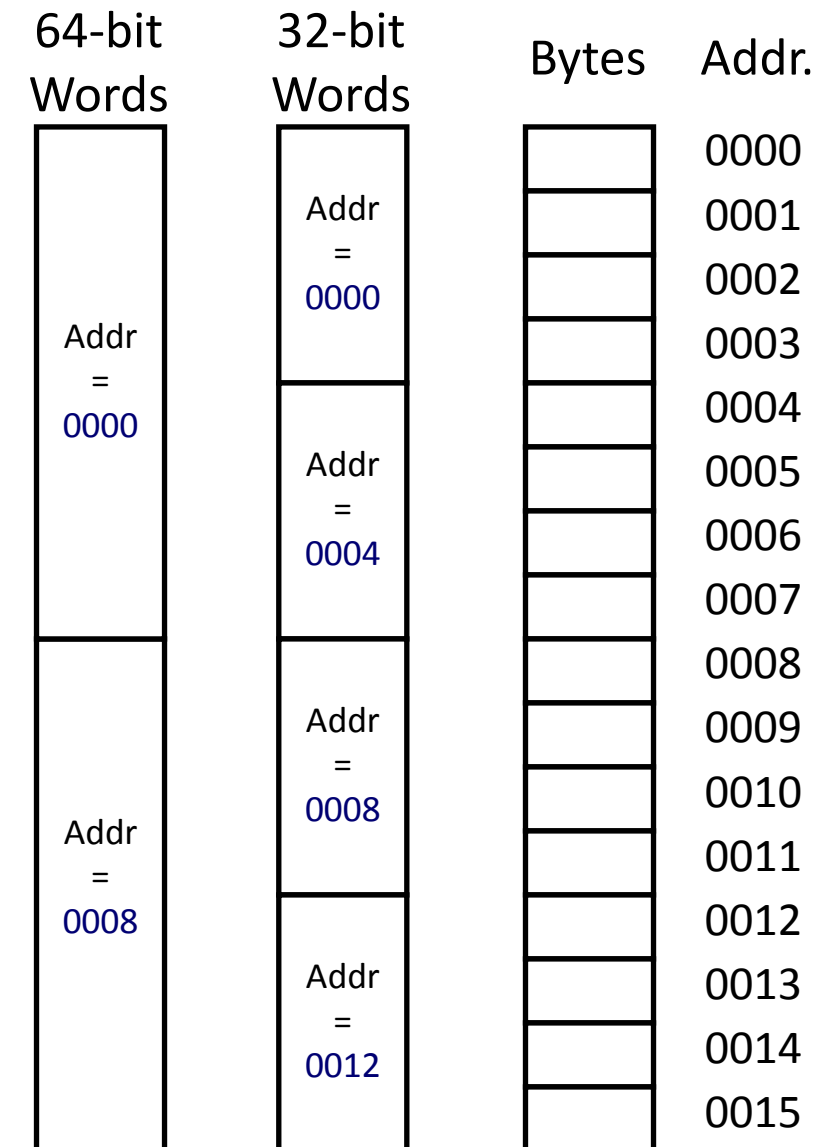
```
Typedef unsigned char bool;
```

```
typedef enum{  
    False=0,  
    True=1  
}bool;
```

- Machine has a “word size”
 - Nominal size of integer-valued data and Including addresses.
- Addresses specify locations of bytes in memory
 - Address of first byte in word
 - Addresses of successive words differ by 4 (32-bit) or 8 (64-bit)



- Machine has a “word size”
 - Nominal size of integer-valued data and Including addresses.
- Addresses specify locations of bytes in memory
 - Address of first byte in word
 - Addresses of successive words differ by 4 (32-bit) or 8 (64-bit)



- Global and static variables.
- Memory is allocated before the execution(run time) of the program. but the values of variables may be changed at run time.
- Variables remain permanently allocated at fixed memory location and freed automatically upon end of the program.
- Note: Local variables created at the start of the block (stored in stack memory) and freed at the end of the block. Some RISC architectures like ARM store the passing parameters of a function and the return address in some CPU internal registers.

- Allocation of memory at the time of execution (run time) is called dynamic memory allocation. It is done using the standard C library functions like **malloc()** and **calloc()**. It is defined in "**stdlib.h**".
- The difference between declaring a normal array and assigning dynamic memory to pointer is the size of an array has to be a constant value which limits its size to what we decide at the moment of designing the program before its execution whereas the dynamic memory allocation allows us to assign memory during the execution of the program(run time) using any variable or constant value as its size.
- The dynamic memory requested by program is allocated by the system in memory **heap**.

▪ malloc()

- The name malloc stands for "memory allocation". The function malloc() reserves a block of memory of specified size and return a pointer of type void which can be casted into pointer of any form. If the space is insufficient, allocation fails and returns NULL pointer.

```
ptr=(cast-type*)malloc(block-size);
```

```
ptr=(int *)malloc(sizeof(int) * 100);
```

- This statement will allocate either 200 or 400 according to size of int 2 or 4 bytes respectively and the pointer points to the address of first byte of memory.
- The allocated memory locations initial values are garbage.

▪ **calloc()**

- The name calloc stands for "contiguous allocation". The only difference between malloc() and calloc() is that, malloc() allocates single block of memory whereas calloc() allocates multiple blocks of memory each of same size and sets all bytes to zero.

```
ptr=(int*)calloc(100,sizeof(int));
```

- This statement allocates contiguous space in memory for an array of 100 elements each of size of int.

▪ **free()**

- Since the necessity of dynamic memory is usually limited to specific moments within a program. Once it is no longer needed it should be freed so that the memory becomes available again for other requests by dynamic memory. We can delete the allocated memory using standard C library function **free()**.

```
free(ptr);
```

▪ **Realloc()**

- If the previously allocated memory is insufficient or more than sufficient. Then you can change memory size previously allocated using **realloc()**.

```
ptr=realloc(ptr,newsize);
```

```
#include <stdio.h>
#include <stdlib.h> //used for malloc & free functions
int main(){
    int n,i,*ptr,sum=0;
    printf("Enter number of elements: ");
    scanf("%d",&n);
    ptr=(int*)malloc(n*sizeof(int)); //memory allocated using malloc
    assert(ptr!=NULL); //check if memory is allocated or not
    printf("Enter elements of array: ");
    for(i=0;i<n;++i) {
        scanf("%d",ptr+i);
        sum+=*(ptr+i); }
    printf("Sum=%d",sum);
    free(ptr); //delete the dynamic allocated memory
    return 0;
}
```

- The computer program memory is organized into the following:
 - **Data Segment** (.data + .bss + Heap + Stack) } **SRAM**
 - **Constant Segment** (.rodata) }
 - **Code segment** (.text) } **Flash EEPROM**

- Mainly Data Segments contain the variables defined in the code, and this segment is initially in RAM

- **.data Segment**

The data area contains global and static variables used by the program that are initialized.

- **.bss Segment**

by default it starts at the end of the data segment. It contains all global variables and static variables that are initialized to zero or do not have explicit initialization in source code.

▪ **Note:**

- Uninitialized static variables always initialized to Zero.
- Uninitialized global variables may initialized to zero or garbage. This can be compiler specific so 'best practice' says that you should not rely on this happening - if you want to assume it has an initial value then you should initialize it by yourself in your source code.

- **.heap Segment**

begins at the end of the .bss segment and grows to larger addresses from there. The Heap area manage dynamic allocation memory requests during run time using standard c libraries like malloc, calloc and free functions.

- **.stack Segment**

traditionally share with the heap the same memory space but it grew in the opposite direction, and it mainly contains the local variables, function passed arguments, function return address.

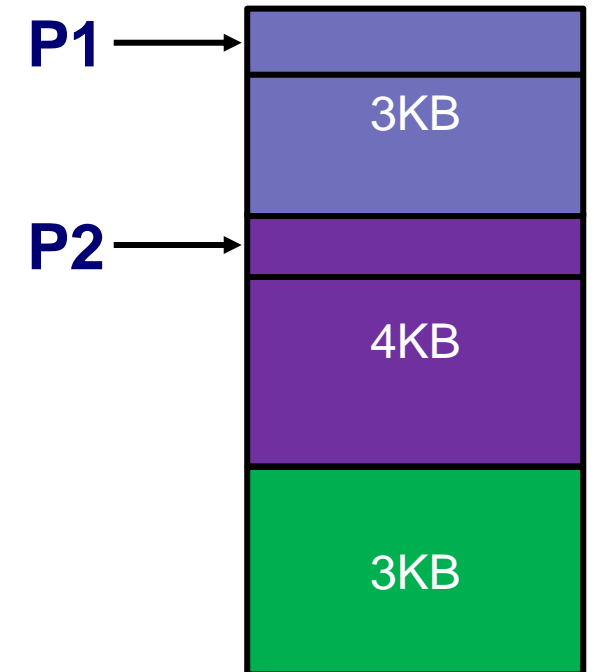
▪ Heap Memory Fragmentation Problem

- The best way to understand memory fragmentation is to look at an example. For this example, it is assumed that there is a 10K heap. First, an area of 3K is requested, thus:

```
#define K (1024)  
char *p1;  
p1 = (char *)malloc(3*K);
```

- Then, a further 4K is requested:

```
p2 = (char *) malloc(4*K);
```



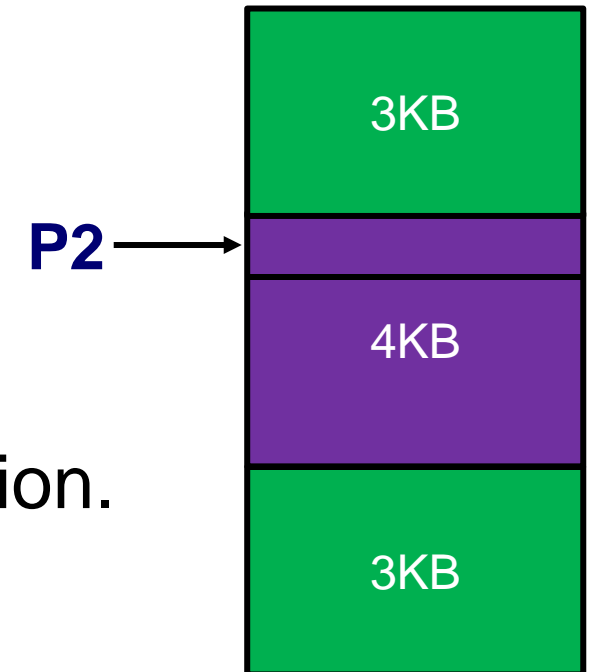
- 3K of memory is now free. Some time later, the first memory allocation, pointed to by p1, is de-allocated:

```
free(p1);
```

- This leaves 6K of memory free in two 3K chunks. A further request for a 4K allocation is issued:

```
p1 = (char *) malloc(4*K);
```

- This results in a failure – NULL is returned into p1 because, even though 6K of memory is available, there is not a 4K contiguous block available. This is memory fragmentation.



- It would seem that an obvious solution would be to de-fragment the memory, merging the two 3K blocks to make a single one of 6K. However, this is not possible because it would entail moving the 4K block to which p2 points. Moving it would change its address, so any code that has taken a copy of the pointer would then be broken.
- There is still a risk that the heap and stack could collide if there are large requirements for either dynamic memory or stack space. The former can even happen if the allocations aren't all that large but dynamic memory allocations get fragmented over time such that new requests don't quite fit into the "holes" of previously freed regions. Large stack space requirements can arise in a C function for large numerous local variables or when use recursive calling function.

- **.text Segment**

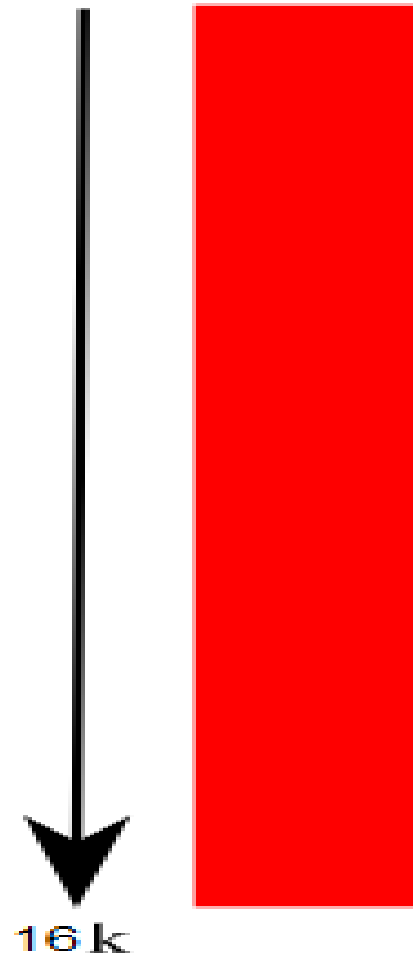
Also called code segment located in Flash EEROM program memory by default and it contains the actual machine code which make up your program. machine code produced from your compiler for the target chip and is 'read-only'.

- **.rodata Segment**

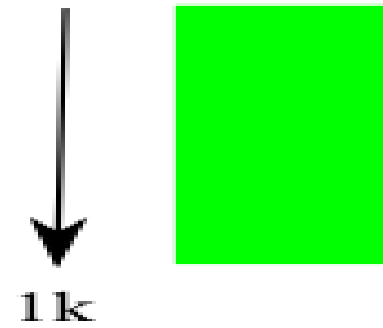
Also called constant segment located in Flash EEROM program memory by default and it contains the constant variables defined by **const** keyword in code, usually located under the code segment.

Available memory types on an ATMega16

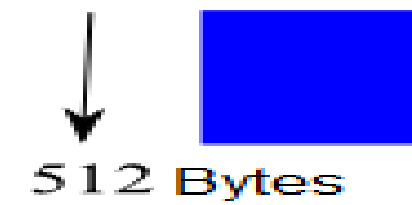
Flash Memory



SRAM



EEPROM



▪ **Flash EEPROM Program Memory**

- It is used to store Read Only information like your program code and constant variables.
- Constant Address and Constant Value allocation(fixed during run time).

▪ **Data Memory Section**

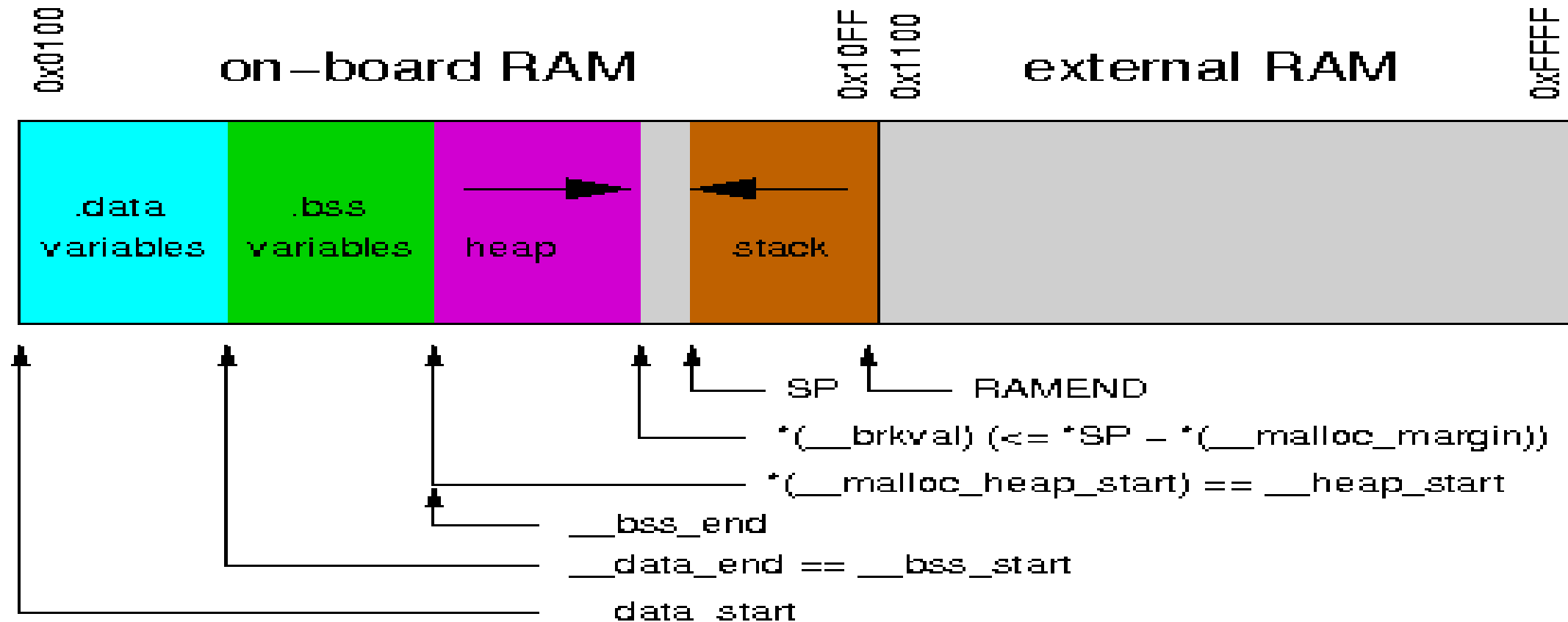
- Located at SRAM Data Memory.
- This is Read Write memory that is used to store changing information.
- Data variables allocated on constant Address but with variable value.

▪ Stack & Heap

- Located at SRAM Data Memory.
- Variables stored at variable locations with variable values.
- Change during run time for stack local variables are allocated at the beginning of function and destroyed at the end of the function. For heap programmer asks for memory allocation using **malloc** and **calloc** functions and programmer freed the allocated space using **free** function.
- If your processor supports a **stack** in general memory, the space required to record the stack is allocated from RAM Data memory that would otherwise be used for global variables.

- Not all stacks are recorded in main (or data) memory: the Microchip PIC and Scenix SX architectures use a stack space outside of user RAM.

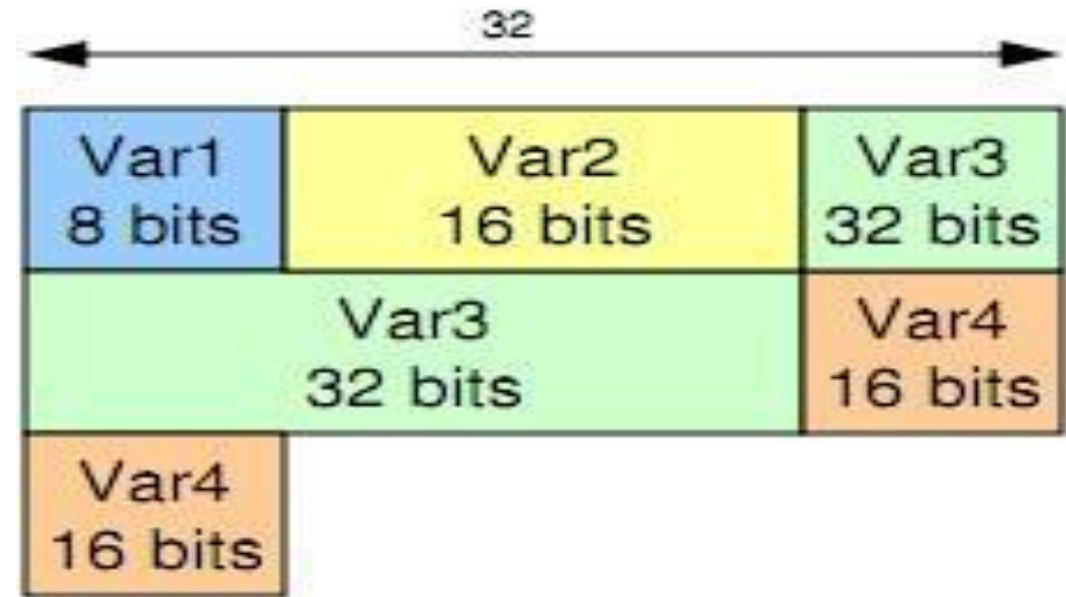
#pragma memory stack [0x40] @ 0xFF; //MC68HC705C8.



AVR Memory

- Consider this structure definition:

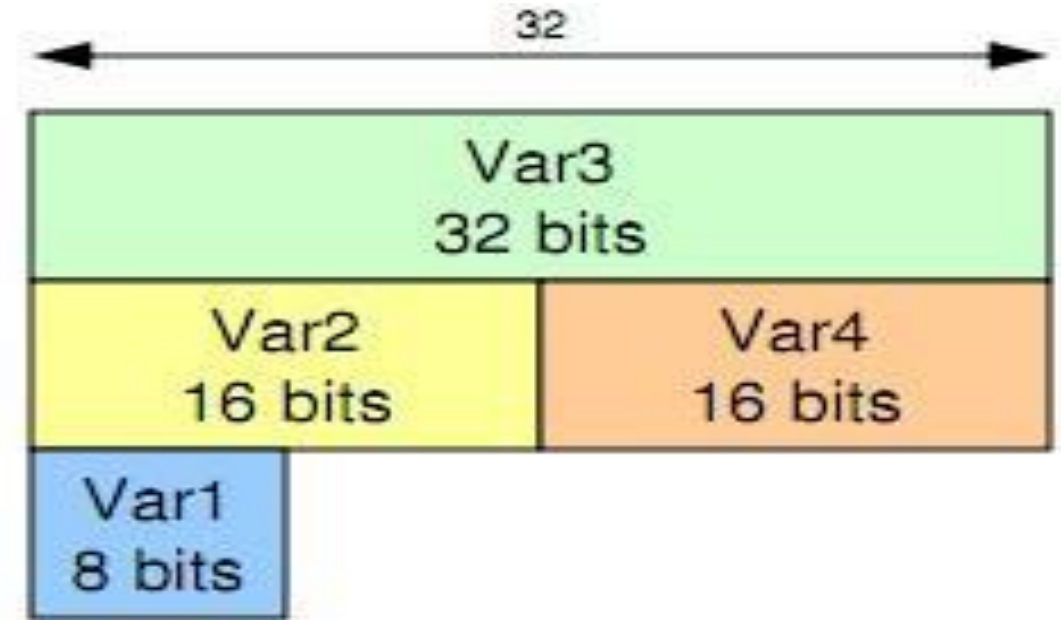
```
typedef struct{  
    byte Var1;  
    word Var2;  
    dword Var3;  
    word Var4;  
}tData;
```



- In a 32 bit address bus architecture, the fetch of Var3 and Var4 require 2 memory fetches to completely get the data of it instead of 1 memory in case of Var1, Var2.

- But if we reorder the elements in the structure definitions like the following:

```
typedef struct{  
    dword Var3;  
    word  Var2;  
    word  Var4;  
    byte  Var1;  
}tData;
```



- In this case the fetch of the 4 variables will cost us only 1 memory fetch !!!!!!!!.
- Memory alignment can be simplified by declaring first the 32-bit variables, then 16-bit, then 8-bit.

- The memory systems on most computers are byte addressable (a unique address for each byte)
 - **Big-endian:**
Store most significant byte in low address and least significant byte in high address (Freescale).
 - **Little-endian:**
Store least significant byte in low address and most significant byte in high address (Intel).
 - **Bi-endian:**
Can be configured to efficiently handle both big and little endian (PowerPC/ARM).

- store the 16-bit number 1000 (0x03E8) at locations 0x2000.0850 and 0x2000.0851.

Address	Data
0x2000.0850	0x03
0x2000.0851	0xE8

Big Endian

Address	Data
0x2000.0850	0xE8
0x2000.0851	0x03

Little Endian

- store the 32-bit number 0x12345678 at locations 0x2000.0850 through 0x2000.0853.

Address	Data
0x2000.0850	0x12
0x2000.0851	0x34
0x2000.0852	0x56
0x2000.0853	0x78

Big Endian

Address	Data
0x2000.0850	0x78
0x2000.0851	0x56
0x2000.0852	0x34
0x2000.0853	0x12

Little Endian

	0x1000	0x1001	0x1002	0x1003	
Big-endian	0x0A	0xC0	0xFF	0xEE	0x0AC0FFEE
Little-endian	0xEE	0xFF	0xC0	0x0A	

Big-endian to Little-endian conversion and back

```
short convert_short(short in)
{
    short out;
    char *p_in = (char *) &in;
    char *p_out = (char *) &out;
    p_out[0] = p_in[1];
    p_out[1] = p_in[0];
    return out;
}
```

```
long convert_long(long in)
{
    long out;
    char *p_in = (char *) &in;
    char *p_out = (char *) &out;
    p_out[0] = p_in[3];
    p_out[1] = p_in[2];
    p_out[2] = p_in[1];
    p_out[3] = p_in[0];
    return out;
}
```

- Set an integer variable at the absolute address 0x67A9 to the value 0xAA55.

```
int *ptr;  
ptr = (int *)0x67A9;  
*ptr = 0xAA55;
```

```
*(int * const)(0x67A9) = 0xAA55;
```

- Also called **Stub** code.
- Startup code is a small block of assembly language code that prepares the hardware to start executing higher level language code (C language).
- Usually startup code is automatically inserted by software development tools.
- Most cross-compilers for embedded systems include an assembly language file called **startup.asm** or **startup.s** or something similar where the startup code is written.
- Architecture and target dependent.

- The startup file containing the code executed before running the software, this code mainly used to initialize and setup the variables to be ready for code execution.
- The main tasks of the startup file are:
 1. Disable all interrupts.
 2. Clock initialization and stabilization.
 3. Copy any initialized data from Flash EEPROM (Program memory) to Data Segments in SRAM(Data memory).
 4. Zero the uninitialized data area in BSS Segment.
 5. Allocate space for and initialize the stack.
 6. Initialize the processor's stack pointer to the end of your SRAM.
 7. Initialize the peripheral registers.
 8. Call **main**.

- **.init0:** Weakly bound to `__init()`. If user defines `__init()`, it will be jumped into immediately after a reset.
- **.init2:** In C programs, weakly bound to initialize the stack, set stack pointer and to clear `__zero_reg__` (r1).
- **.init4:** For devices with > 64 KB of ROM, .init4 defines the code which takes care of copying the contents of .data from the flash to SRAM. For all other devices, this code as well as the code to zero out the .bss section is loaded from libgcc.a.
- **.init6:** Unused for C programs, but used for constructors in C++ programs.
- **.init9:** Jumps into `main()`.
- Others is user defined and can edit in it.

```
#include <avr/io.h>
```

```
; The ,"ax",@progbits tells the assembler that the section is allocatable ("a"),  
; executable ("x")  
; contains data ("@progbits").
```

```
.section .init1,"ax",@progbits  
ldi r0, 0xff  
out _SFR_IO_ADDR(PORTB), r0  
out _SFR_IO_ADDR(DDRB), r0
```

```
#include <avr/io.h>
```

```
void my_init_portb(void) __attribute__((naked)) \  
__attribute__((section (".init3")));
```

```
void my_init_portb(void)  
{  
    PORTB = 0xff;  
    DDRB = 0xff;  
}
```

- These sections are used to define the exit code executed after return from `main()` or a call to **`exit()`**. These all are subparts of the **`.text` section**.
- The **`.finiN`** sections are executed in descending order from 9 to 0.
 - **`.fini9`**: Unused. User definable. This is effectively where `_exit()` starts.
 - **`.fini6`**: Unused for C programs, but used for destructors in C++ programs.
 - **`.fini0`**: Goes into an infinite loop after program termination and completion of any `_exit()` code (execution of code in the `.fini9` → `.fini1` sections).
 - Others is user defined and can edit in it

- Modular programming is a software design technique that emphasizes separating the functionality of a program into independent, interchangeable modules, such that each contains everything necessary to execute only one aspect of the desired functionality.
- Conceptually, modules represent a separation of concerns, and improve maintainability by enforcing logical boundaries between components.
- Modules are typically incorporated into the program through interfaces. A module interface expresses the elements that are provided and required by the module.
- The elements defined in the interface are detectable by other modules. The implementation contains the working code that corresponds to the elements declared in the interface

▪ Advantages

- Several programmers can work on individual programs at the same time, thus, making development of program faster. The code base is easier to debug, update and modify. It leads to a structured approach as a complex problem can be broken into simpler tasks.
- With modular programming, concerns are separated such that modules perform logically discrete functions. No (or few) modules interact with other modules of the system; except in the sense that one module may use another module, to achieve its purpose.

The desired module goal is to have no interaction between modules(independent and abstracted Module).

- **Header** files are files that are included in other files prior to compilation by the C preprocessor. Some, such as **stdio.h**, are defined at the system level and must be included by any program using the standard I/O library.
- **Header** files are also used to contain data declarations and defines that are needed by more than one program.
- **Header** files should be functionally organized, i.e., declarations for separate subsystems or functions should be in separate header files.
- Also, if a set of declarations is likely to change when code is ported from one machine to another, those declarations should be in a separate header file.

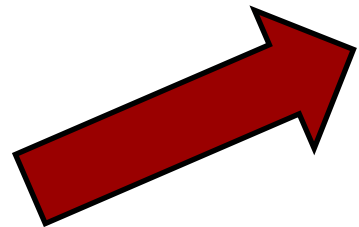
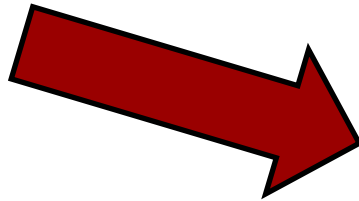
file1.h

```
int add(int,int)
```

file2.h

```
#include "file1.h"
```

```
int sub(int,int());
```



main.c

```
#include "file1.h"  
#include "file2.h"
```

```
int main() {  
    /* code */  
    return 0;  
}
```

file1.h
included
twice


- Header files should not be nested. The prologue for a header file should, therefore, describe what other headers need to be #included for the header to be functional. In extreme cases, where a large number of header files are to be included in several different source files, it is acceptable to put all common #includes in one include file.
- It is common to put the following into each .h file to prevent accidental double-inclusion.

```
#ifndef EXAMPLE_H  
#define EXAMPLE_H  
    /* body of example.h file */  
#endif /* EXAMPLE_H */
```

This double-inclusion mechanism should not be relied upon, particularly to perform nested includes.

- Large projects may potentially involve many hundreds of source files (modules).
- Global variables and functions in one module may be accessed in other modules.
- Global variables and functions may be specifically hidden inside a module.
- Maintaining consistency between files can be a problem.

■ Data Sharing



```
extern float step;

void print_table(double, float);

int main(void)
{
    step = 0.15F;

    print_table(0.0, 5.5F);

    return 0;
}
```

```
#include <stdio.h>

float step;

void print_table(double start, float stop)
{
    printf("Celsius\tFahrenheit\n");
    for(;start < stop; start += step)
        printf("%.11f\t%.11f\n", start,
                start * 1.8 + 32);
}
```

■ Data Hiding

When **static** is placed before a global variable, or function, the item is locked into the module

```
static int entries[S_SIZE];
static int current;

void push(int value)
{
    entries[current++] = value
}

int pop(void)
{
    return entries[--current];
}

static void print(void)
{
}

void push(int value);
int pop(void);
void print(void);
extern int entries[];

int main(void)
{
    push(10); push(15);
    printf("%i\n", pop());

    entries[3] = 77;
    print();
    return 0;
}
```

```
extern float step;

void print_table(double, float);

int main(void)
{
    step = 0.15F;

    print_table(0.0, 5.5F);

    return 0;
}
```

Don't Do This

```
#include <stdio.h>

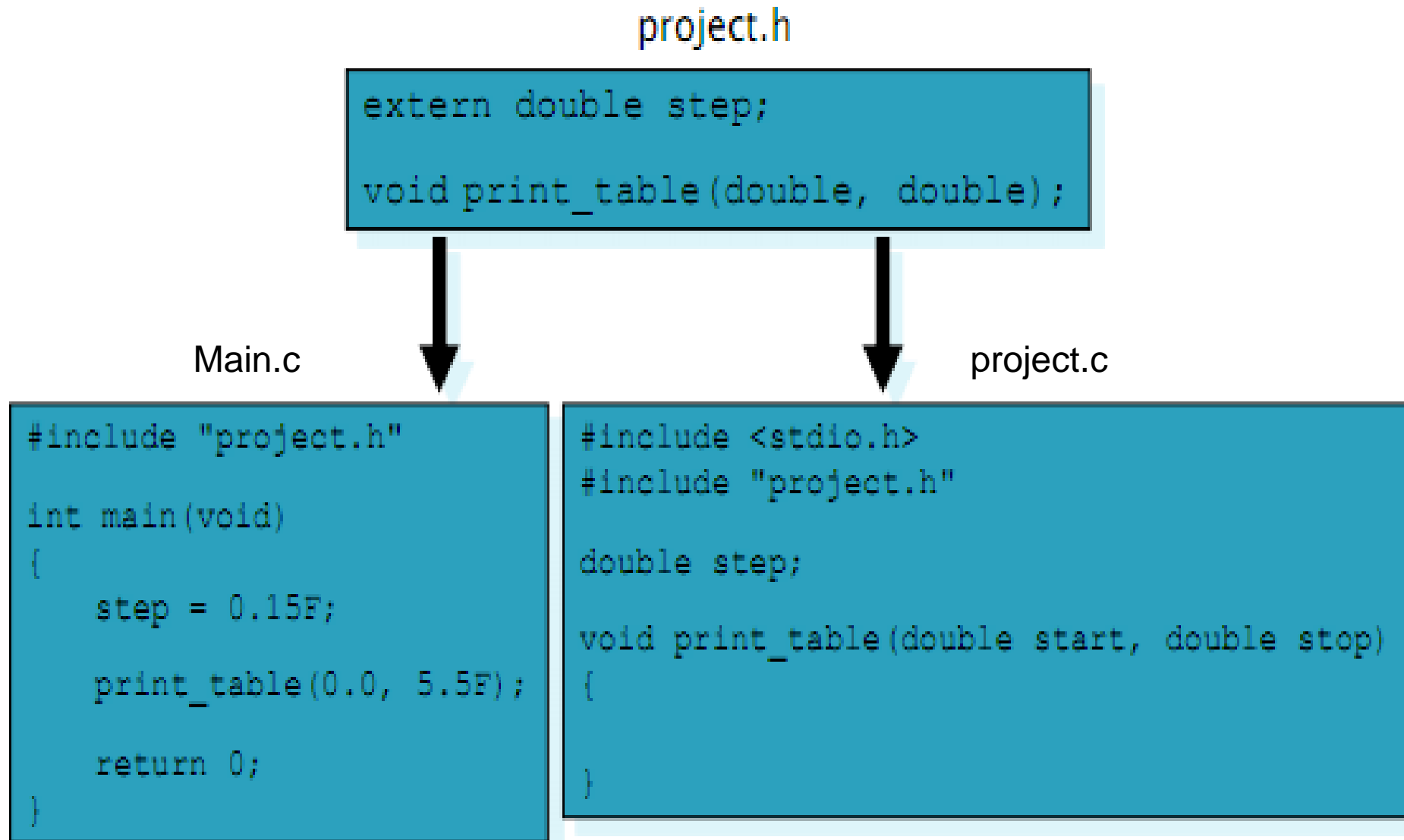
float step;

void print_table(double start, float stop)
{
    printf("Celsius\tFahrenheit\n");
    for(; start < stop; start += step)
        printf("%.11f\t%.11f\n", start,
                start * 1.8 + 32);
}
```

■ Use Header Files

- Maintain consistency between modules by using header files.
- NEVER place an extern declaration in a module.c file.
- NEVER place a prototype of a non static (i.e. Sharable) function in a module.c file.

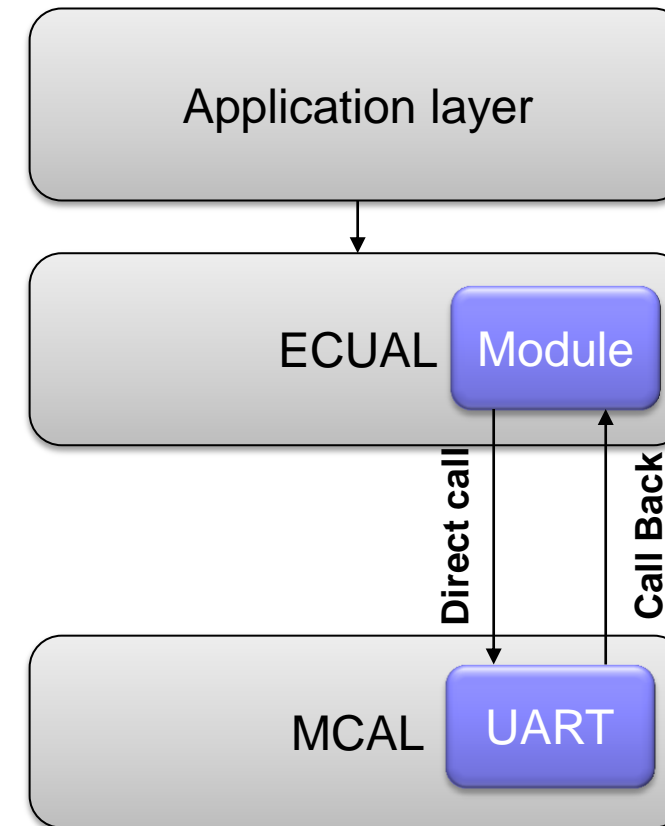
Include files correctly



- A function pointer is a variable that stores the address of a function that can later be called through that function pointer.
- a function name is really the starting address in memory of the code that performs the function's task. Pointers to functions can be passed to functions, returned from functions, stored in arrays and assigned to other function pointers.

- Definition:
Return_Value (*ptr_name)(type argument_name);
- Initialization:
ptr_name=func_name;
- Calling:
same as function calling

- The Call back function is just a normal calling using a pointer to function.
- Upper layer module functions can call the lower layer module functions using a direct function call (API).
- It is always used to make a needed call from a lower layer to upper layer like calling a function from the scheduler component by timer component to provide it with the needed configurations.



```
//call_back_ptr must be global and volatile
volatile void (*call_back_ptr)(unsigned char)=NULL;

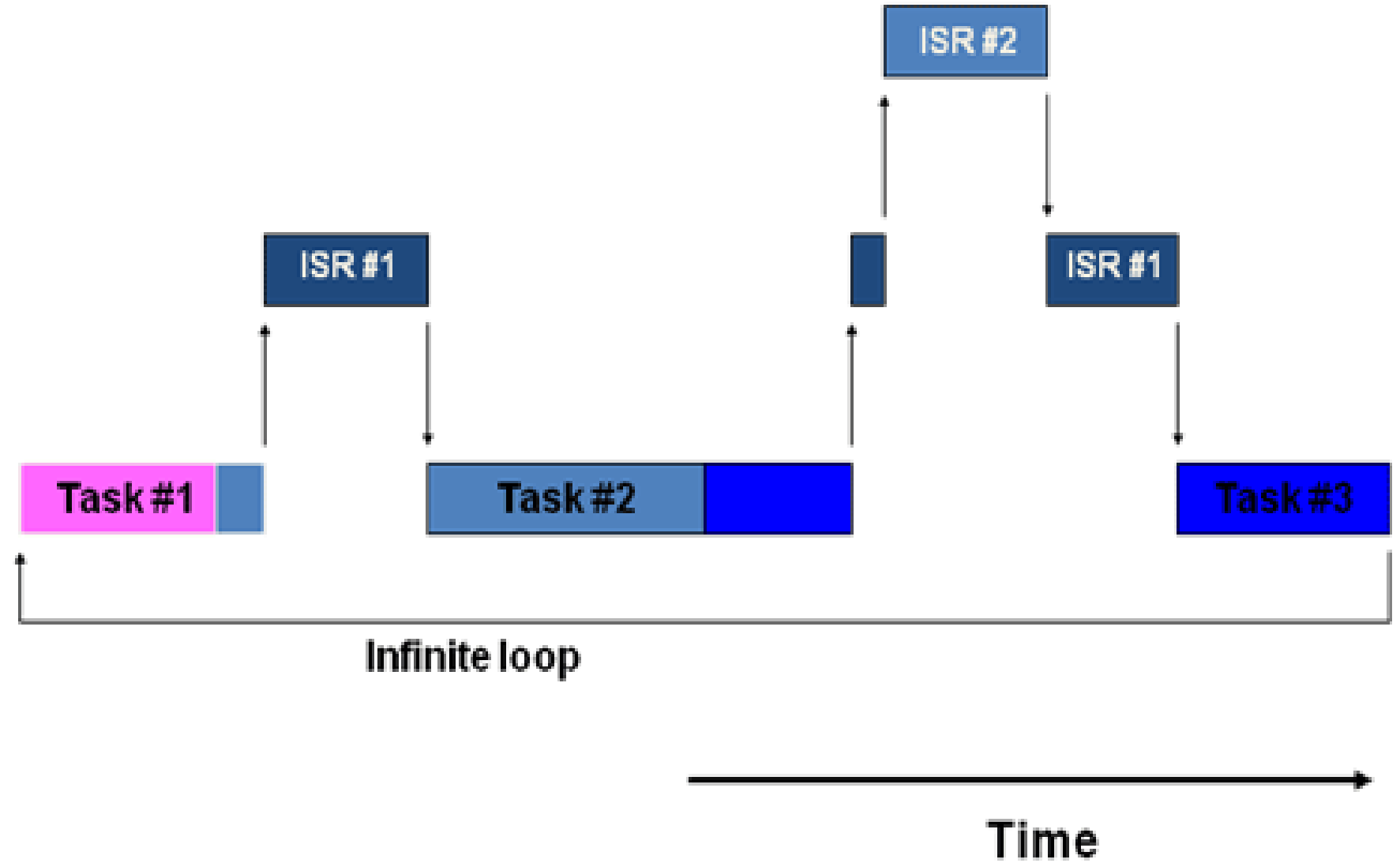
ISR(RXC_vect){ //interrupt when UART receive any data
    if(call_back_ptr!=NULL){
        /*call function in upper layer module and pass the received data*/
        (*call_back_ptr)(UDR);
    }
}

Void uart_init(unsigned char baud_rate, void (*func_ptr)(unsigned char)){
    /* UART initialization code*/
    call_back_ptr=func_ptr;
}
```

Foreground #2

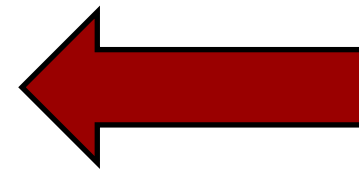
Foreground #1

Background



```
/* Background */  
void main (void)  
{  
    Initialization;  
    FOREVER  
    {  
        Read analog inputs;  
        Read discrete inputs;  
        Perform monitoring functions;  
        Perform control functions;  
        Update analog outputs;  
        Update discrete outputs;  
        Scan keyboard;  
        Handle user interface;  
        Update display;  
        Handle communication requests;  
        Other...  
    }  
}
```

**Interrupt
request**



End of ISR

```
/* Foreground */  
ISR (void)  
{  
    Handle asynchronous event;  
}
```

- **Synchronous functions**

will return control to caller only after finishing its task.

- **Asynchronous functions**

will just start its task and return control to caller, while its task is continue in the background.

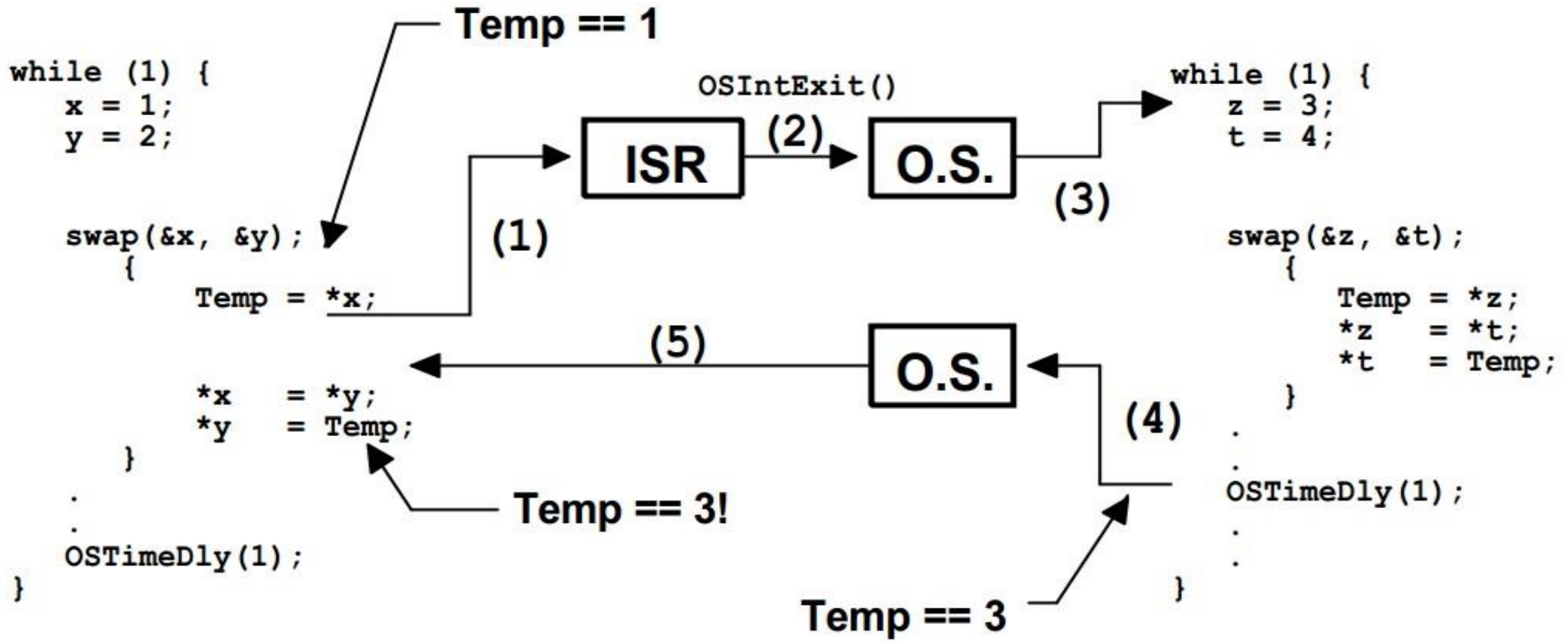
- A reentrant function is a function that can be used by more than one task without fear of data corruption.
- A reentrant function can be interrupted at any time and resumed at a later time without loss of data or data corruption.
- An error caused by a non-reentrant function may not show up in your application during the testing phase.


```
int Temp;  
  
void swap(int *x, int *y)  
{  
    Temp = *x;  
    *x    = *y;  
    *y    = Temp;  
}
```

Non-Reentrant Function

LOW PRIORITY TASK

HIGH PRIORITY TASK



- Temp variable here is a shared resource. So exclusive access must be guaranteed.
- We can make swap() reentrant by using one of the following techniques:
 - Declare Temp local to swap().
 - Disable interrupts before the operation and enable them after.
 - Use a semaphore.

▪ **Function to be Reentrant**

- Must hold no static or global non-constant data.
- Must work only on the data provided to it by the caller and local variables.
- Ensure mutual exclusion of shared hardware and software resources.
- Must not call non-reentrant computer programs or routines.

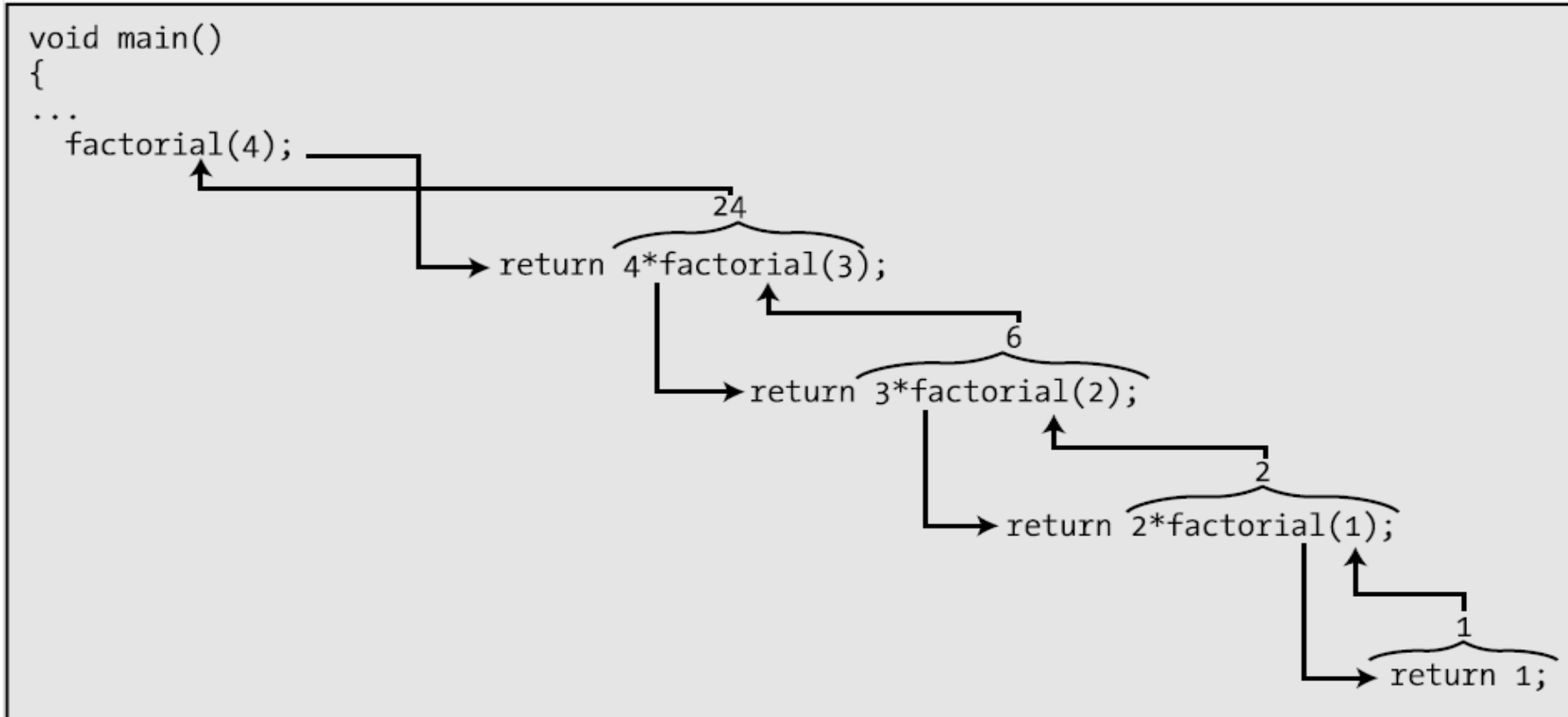
▪ **The following functions must be reentrant**

- Functions shared between different tasks in a multitasking system.
- Functions shared between ISR and background task.
- Function shared between different ISRs with different priorities, when nesting is enabled.

```
void strcpy(char *dest, char *src)
{
    while (*dest++ = *src++) {
        ;
    }
    *dest = NUL;
}
```

Reentrant Function

- Recursive function is function that call itself.
- Recursion is not recommended in embedded software:
 - High stack consumption, because each call requires allocation of a new copy of local variables.
 - The execution time needed for the allocation and de-allocation, and for the storage of those parameters and local variables can be costly.
 - The recursive code is hard to test and hard to read.
 - Precautions need to be taken to ensure that the recursive routine will terminate, otherwise the run-time stack will overflow.



- A Critical section of code ,also called critical region is a piece of code that accesses a shared resource (data structure or device) that must not be concurrently accessed by more than one thread(Task) of execution..
- A critical section is code that once starts executing, it must not be interrupted to ensure this, interrupts are typically disabled before the critical code is executed and enabled when the critical code is finished.

```
Disable interrupts;  
Access the resource (read/write from/to variables);  
Reenable interrupts;
```


- Timeout mechanism is used to release the CPU while waiting for an event when this event does not triggered within the specified timeout period.
- To implement Timeout:
 - Timeout period is determined from the nature of the event.
 - Timeout can be implemented by hardware timer or by software depending on available resources and needed accuracy.
 - Actions to be performed when timeout expired

▪ Simulator

- used to simulate the behavior of the hardware without the existence of the hardware itself.
- It is a software on the PC in order to expect the behavior of the target μC .
- The simulator can understand the assembly instruction of the μC and it acts as if it was the μC and start changing the contents of RAM, Registers according to the code.
- The simulator can't simulate the real time property of the embedded system.

- Simulator can be used to calculate the time consumed by a part of code by calculating the number of cycles consumed to execute this code, and using the info of the frequency of the system, we could calculate a expected time for the code.

▪ Emulator

- a certain hardware for a specific family of controllers.
- It contains the max RAM & ROM size and all the peripherals and registers that could exist in a microcontroller in this family.
- It is FPGA that implement the core of the μ C.
- It permits the programmer to make a hardware breakpoint and halt the hardware registers, so the programmer simply could debug the constraints in real time manners.

▪ **In Circuit Emulator**

- Simply it is the emulator connected to the board as if it is the μC itself and at this time I could emulate the I/O ports and other functionalities in my system.
- It is done by make the board without the MC then connect the emulator in the μC place.

▪ **Programmer/Flasher**

- It is hardware used in order to burn the code to the Flash EEPROM of the μC .





Contact Details

Eng. Mohamed Tarek.
Embedded Software Engineer
Mob: 01115154316
mtarek.2013@gmail.com