# Embedded Driver Development

Day 1 & 2

# Topics to be discussed

- Introduction to Embedded Programming
- Getting to know the Embedded Target
- Datatypes and Variables
- Address
- Storage Classes
- Function

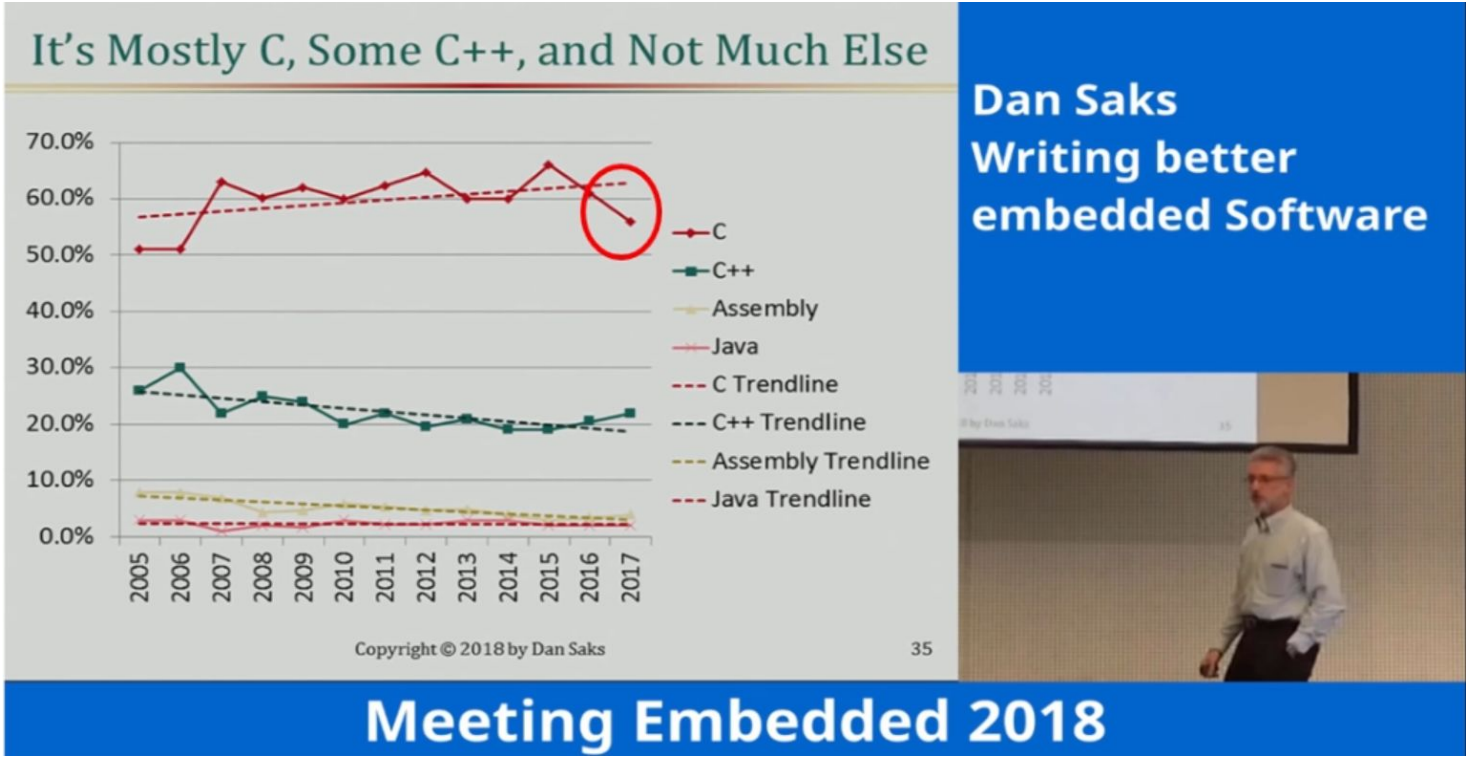
Program???

What is there in a program??

# Popular Programming Languages used for Embedded System

- C & C++ ( Widely Used)
- Rust
- Assembly
- Java
- Python

# Survey by Embedded.com



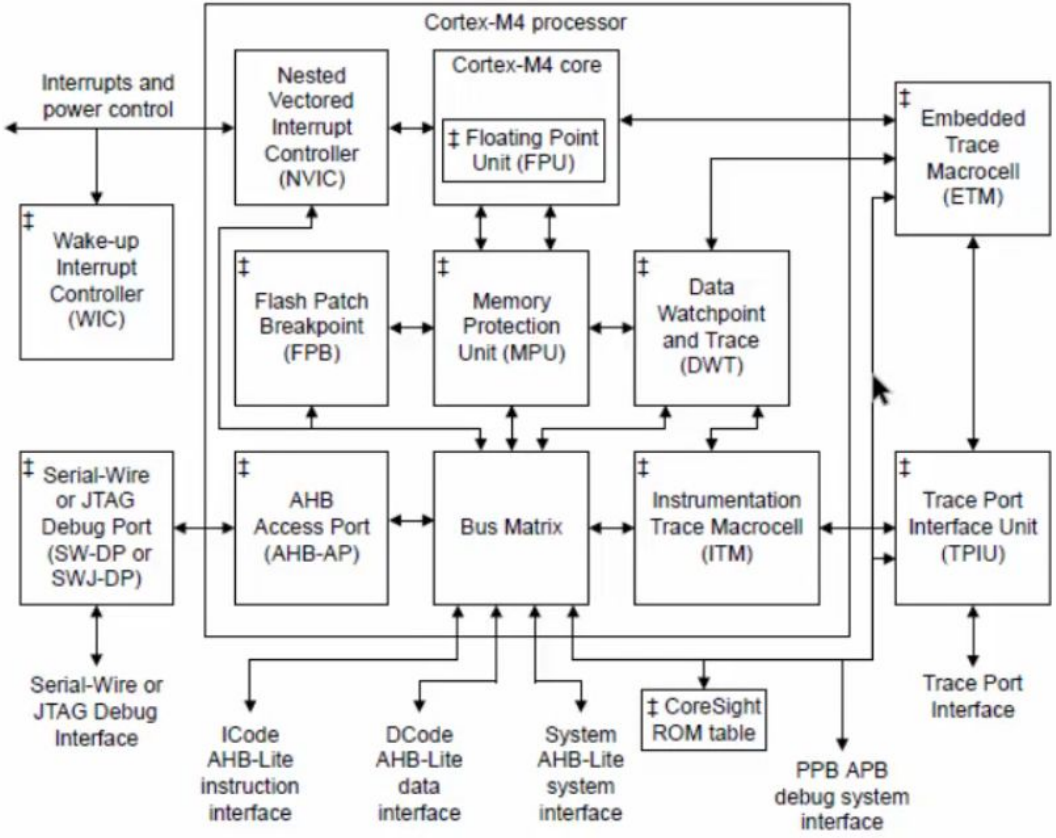
# Embedded Target Introduction

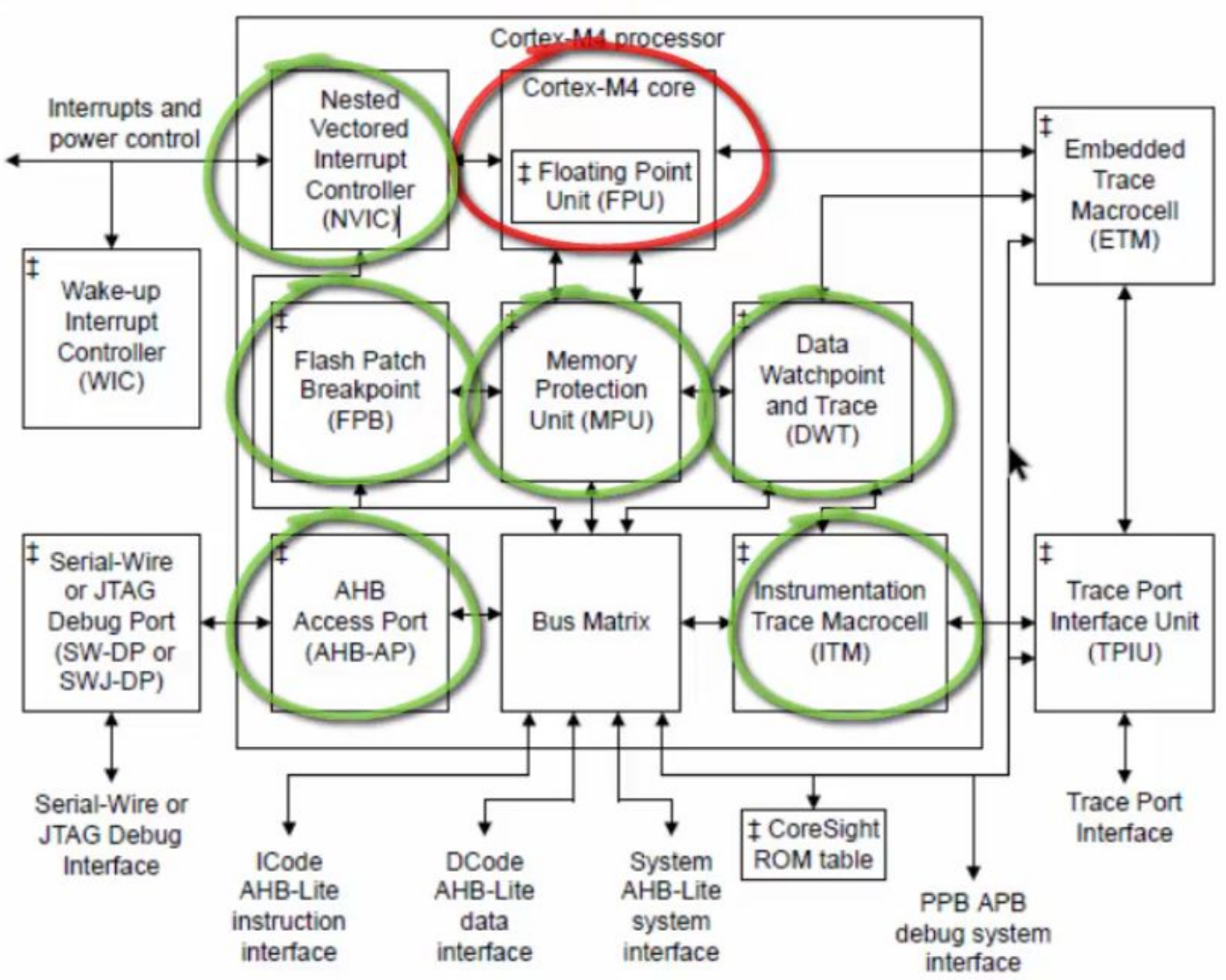
https://www.st.com/en/evaluation-tools/stm32f4discovery.html

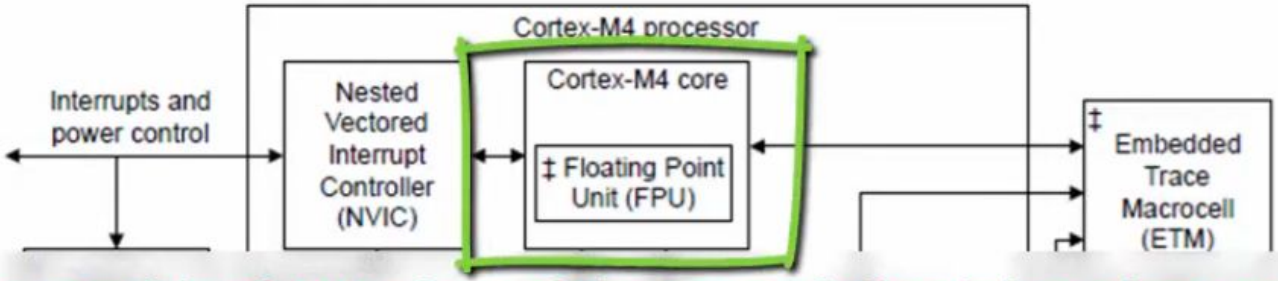
# Processor and Processor Core

Arm Cortex M4 technical reference manual :
https://developer.arm.com/documentation/100166/0001

# Block Diagram




Cortex-M4 processor
Interrupts and power control
Nested Vectored Interrupt Controller (NVIC)
Cortex-M4 core
‡ Floating Point Unit (FPU)
‡ Embedded Trace Macrocell (ETM)
‡ Wake-up Interrupt Controller (WIC)
‡ Flash Patch Breakpoint (FPB)
‡ Memory Protection Unit (MPU)
‡ Data Watchpoint and Trace (DWT)
‡ Serial-Wire or JTAG Debug Port (SW-DP or SWJ-DP)
‡ AHB Access Port (AHB-AP)
Bus Matrix
‡ Instrumentation Trace Macrocell (ITM)
‡ Trace Port Interface Unit (TPIU)
Serial-Wire or JTAG Debug Interface
ICode AHB-Lite instruction interface
DCode AHB-Lite data interface
System AHB-Lite system interface
‡ CoreSight ROM table
PPB APB debug system interface
Trace Port Interface




Core consists of ALU where data computation takes place and result will be generated

It has the logic to decode and execute an Instruction

It has many registers to store and manipulate data

It has pipe line engine to boost the instruction execution

It consists of hardware multiplication and division engine

Address generation unit

# Online GDB

https://www.onlinegdb.com/

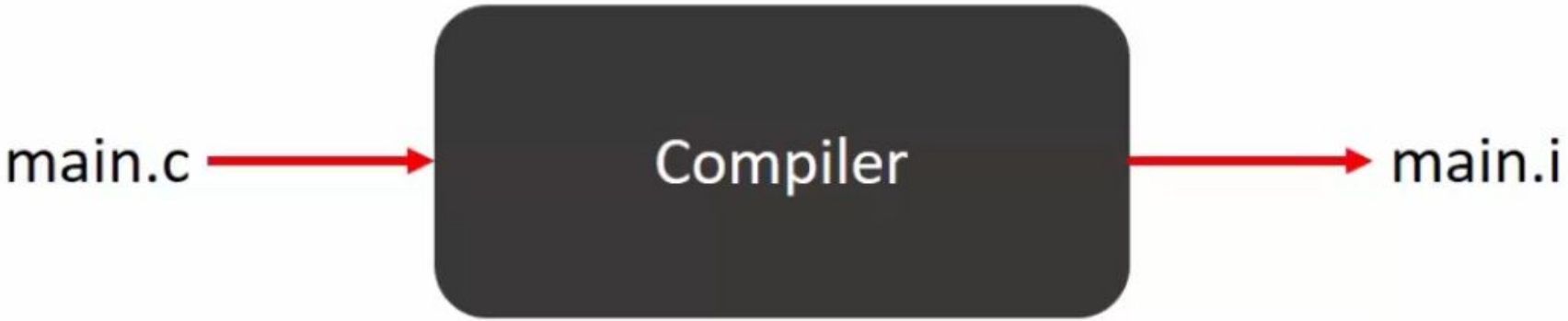
# First ‘C’ Program “Hello World”

Let’s write a ‘C’ code which simply displays the text Hello world on the “console” and exits.

Modify the settings of the Compiler

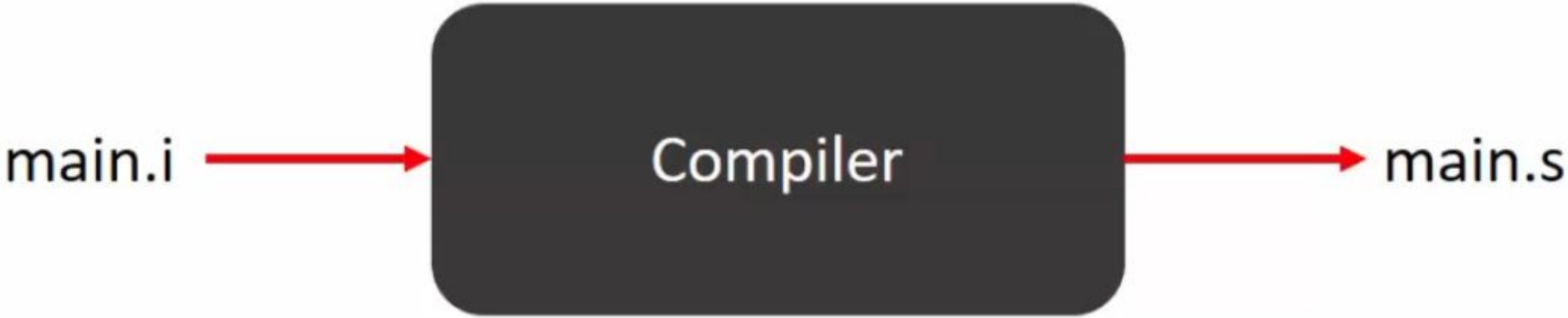
**Type the compiler Argument :  -save-temps**

**Rerun the code**



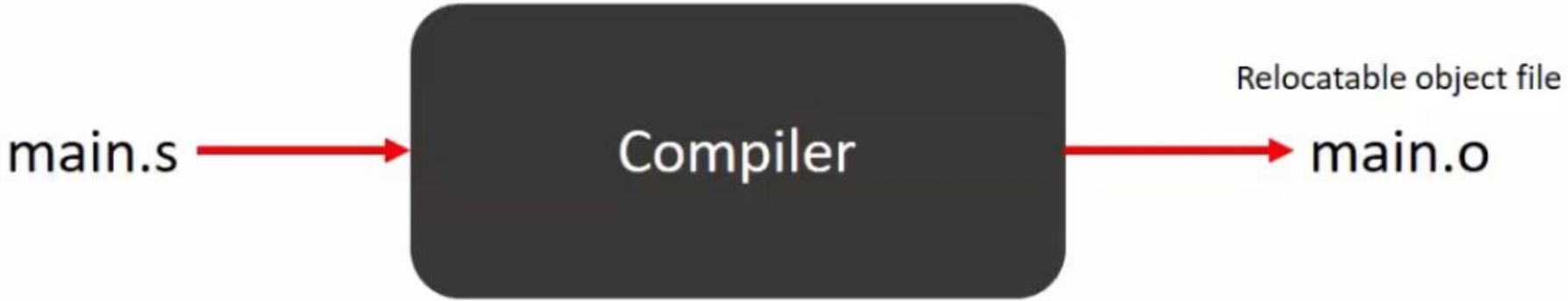

(Pre-processing stage)

All pre-processing directives will be resolved




(Code generation stage)

Higher level language code statements will be converted into processor architectural level mnemonics




(Assembler stage)

Assembly level mnemonics are converted into opcodes(machine codes for instructions)

# Assignment 1: (In Class)

**WAP to print the following in a certain way mentioned below**

**Hello World!!**

**I am enjoying the day!!**

# ‘C’ data types and variables

# Data types :

✓Data type is used for declaring the type of a variable

✓In C programming, data types determine the type and size of data associated with variables

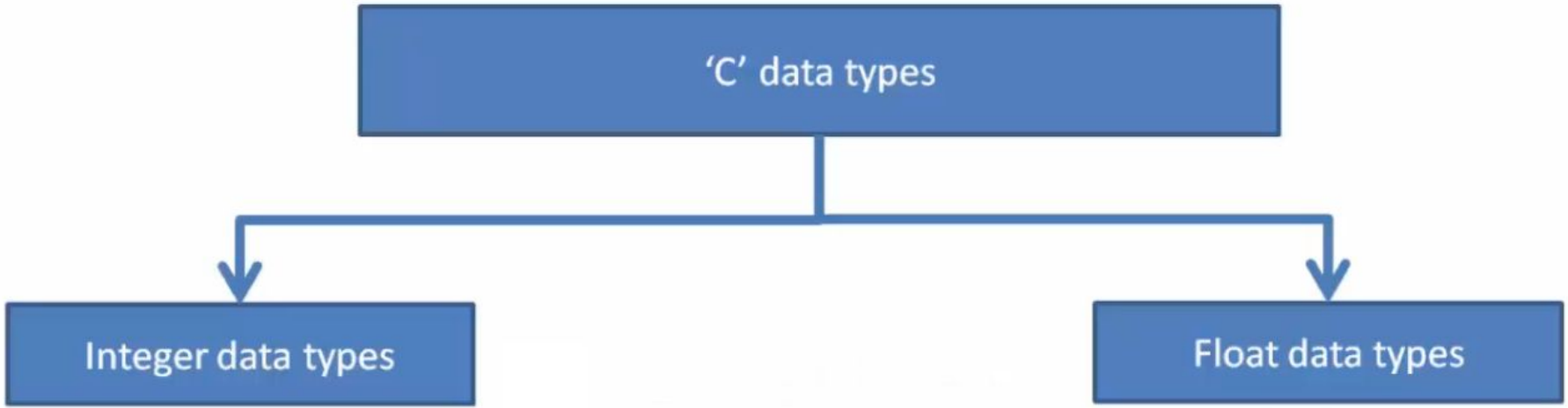
✓Before storing any value in a variable, first programmer should decide its type

# Representing real world data

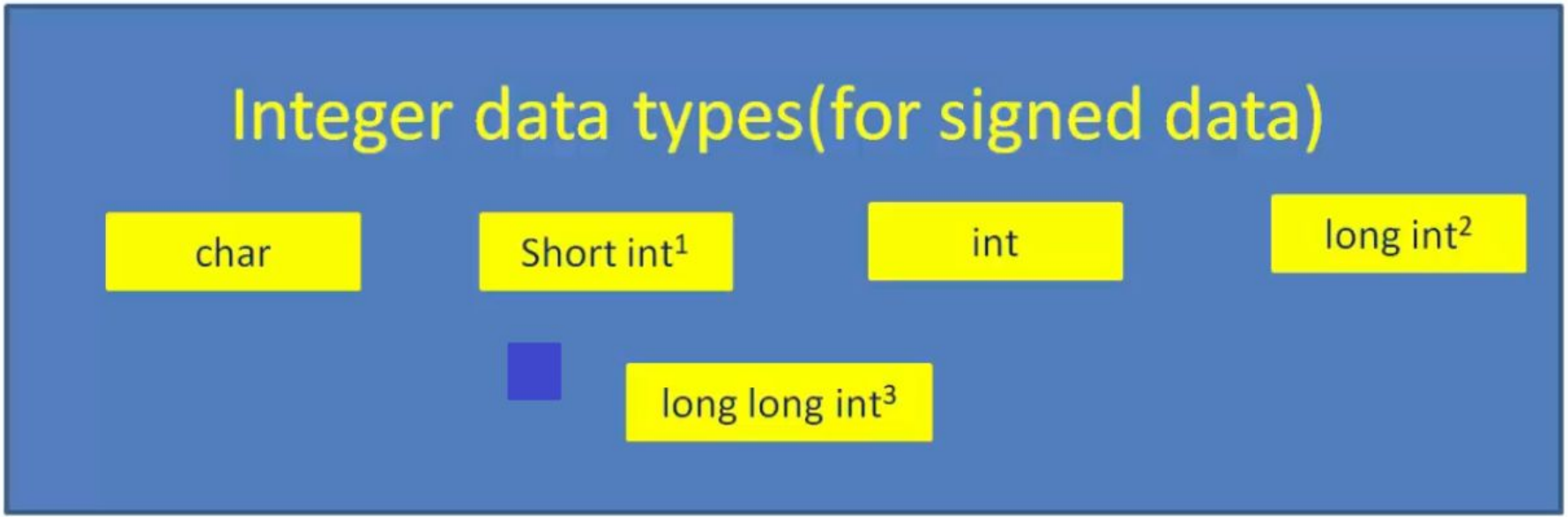
- Data as numbers (Integer or real numbers )
- Data as characters
- Data as strings (Collection of characters )

# Examples

- Roy's age is 44 years, 6 months and 200 days
- The temperature of the city X is +27.2 degrees Celsius.
- Today date is 21st June
- I got an 'A' grade in school assignment


'C' data types
Integer data types
Float data types

Integer data are represented by different integer data types




1: also referred by just “short”; “int” is assumed
‘short int’ is synonymous with ‘short’
2 : also referred by just "long"; "int" is assumed
3: also referred by just "long long"; "int" is assumed

# Integer data types(for unsigned data)

- unsigned char
- unsigned short int
- unsigned int
- unsigned long int
- unsigned long long int

# ‘C’ integer data types , their storage sizes and value ranges

| DATA TYPE | MEMORY SIZE (BYTES) | RANGE |
|---|---|---|
| signed char | 1 | -128 to 127 |
| unsigned char | 1 | 0 to 255 |
| short int | 2 | -32,768 to 32,767 |
| unsigned short int | 2 | 0 to 65,535 |
| int | 4 | -2,147,483,648 to 2,147,483,647 |
| unsigned int | 4 | 0 to 4,294,967,295 |
| long | 8 | -9223372036854775808 to 9223372036854775807 |
| unsigned long | 8 | 0 to 18446744073709551615 |
| long long int | 8 | -9223372036854775808 to 9223372036854775807 |
| unsigned long long int | 8 | 0 to 18446744073709551615 |

**Meaning of memory size :**
**The compiler(e.g., GCC) will generate the code to allocate 64 bits (8 bytes of memory) for each long long variable.**

## XC8 is a cross compiler for PIC 8 bit microcontrollers

| Data Type | Size in Bits | Data Range/Usage |
|---|---|---|
| unsigned char | 8-bit | 0 to 255 |
| char | 8-bit | –128 to +127 |
| unsigned int | 16-bit | 0 to 65,535 |
| int | 16-bit | –32,768 to +32,767 |
| unsigned short | 16-bit | 0 to 65,535 |
| short | 16-bit | –32,768 to +32,767 |
| unsigned short long | 24-bit | 0 to 16,777,215 |
| short long | 24-bit | –8,388,608 to +8,388,607 |
| unsigned long | 32-bit | 0 to 4,294,967,295 |
| long | 32-bit | –2,147,483,648 to +2,147,483,648 |

# armcc cross compiler for 32 bit ARM based microcontrollers

| Type | Size in bits | Natural alignment in bytes | Range of values |
|---|---|---|---|
| char | 8 | 1 (byte-aligned) | 0 to 255 (unsigned) by default.<br>–128 to 127 (signed) when compiled with `--signed_chars`. |
| signed char | 8 | 1 (byte-aligned) | –128 to 127 |
| unsigned char | 8 | 1 (byte-aligned) | 0 to 255 |
| (signed) short | 16 | 2 (halfword-aligned) | –32,768 to 32,767 |
| unsigned short | 16 | 2 (halfword-aligned) | 0 to 65,535 |
| (signed) int | 32 | 4 (word-aligned) | –2,147,483,648 to 2,147,483,647 |
| unsigned int | 32 | 4 (word-aligned) | 0 to 4,294,967,295 |
| (signed) long | 32 | 4 (word-aligned) | –2,147,483,648 to 2,147,483,647 |
| unsigned long | 32 | 4 (word-aligned) | 0 to 4,294,967,295 |
| (signed) long long | 64 | 8 (doubleword-aligned) | –9,223,372,036,854,775,808 to 9,223,372,036,854,775,807 |
| unsigned long long | 64 | 8 (doubleword-aligned) | 0 to 18,446,744,073,709,551,615 |

# Integer data type : char

✓This is an integer data type to store a single character(ASCII code) value or 1 byte of signed integer value (+ve or –ve value)

✓A char data type variable consumes 1 byte of memory.

✓char happens to be the smallest integer data type of 1 byte.

✓There is no other special meaning for the char data type, and it is just another integer data type.

# Range of char data type

??

# Brain Teaser!!!!!

I want to store the current temperature data of city ‘X’ in my program.

City X's today's temperature is 25 degree Celsius.

I am sure that X's temperature never goes below 0 degrees and never goes above 40 degrees Celsius.

That means City X's temperature will always be a positive value, and the max value is less than 255.

# Variable definition

Data type Variable name

```
unsigned char cityXTemperature;    /* This is a variable definition */

cityXTemperature = 25;             /* This is variable initialization */
```

# Assignment 2: (In Class)

Write a C program to calculate and print distance from A to C

# STEPS

1. Create the variables to hold these data
2. Compute the sum of the data
3. Store the result
4. Print the result as shown in the format

# sizeof operator

- sizeof operator of C programming language is used to find out the size of a variable.
- The output of the sizeof operator may be different on different machines because it is compiler dependent.

# Variable scopes

- Variables have scopes
- A **Variable scope** refers to the accessibility of a variable in a given program or function
- For example, a variable may only be available within a specific function, or it may be available to the entire C program

# Variable scopes

- Local scope variables  (local variables )
- Global scope variables ( global variables )

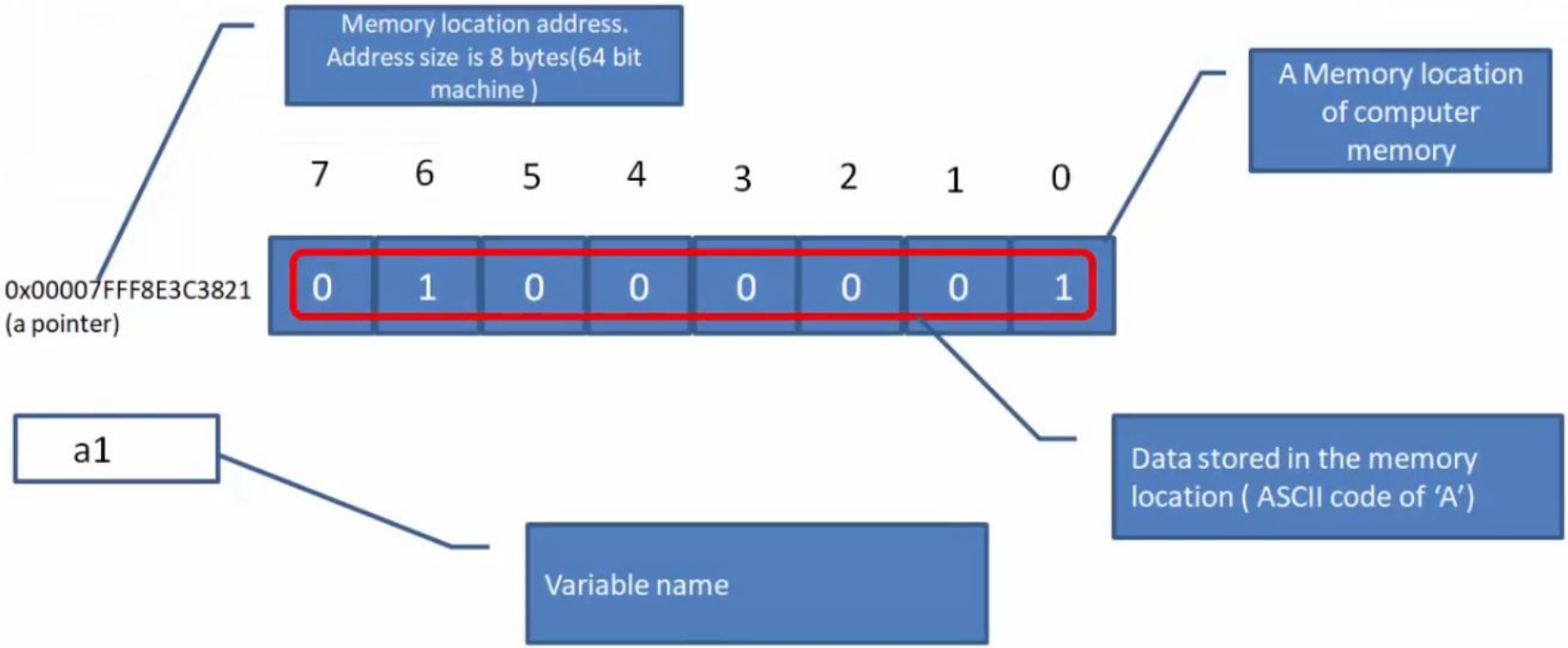
Always remember that when the execution control goes out of the scope of a local variable, the local variable dies that means a variable loses its existence

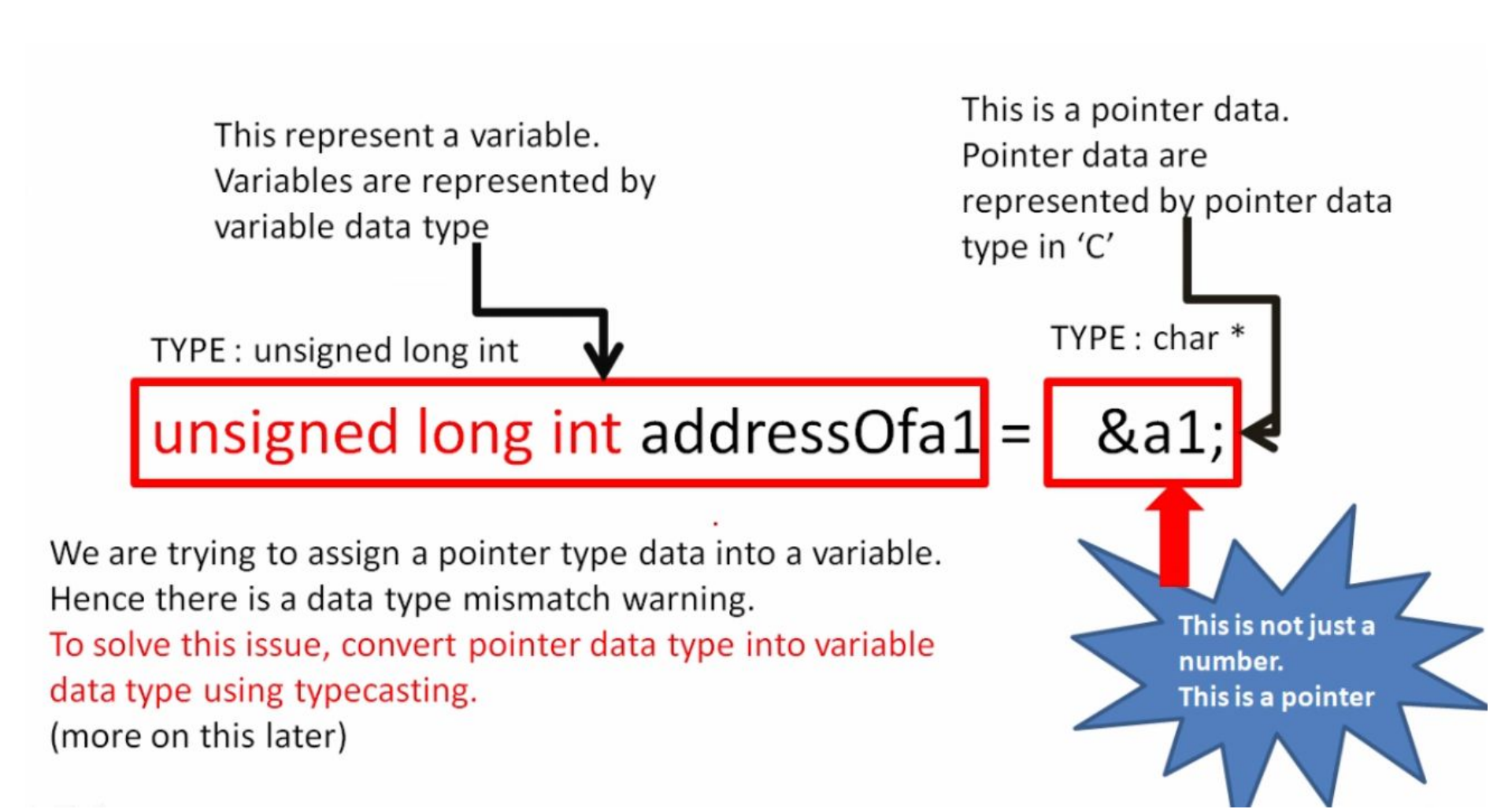
# Address of a variable

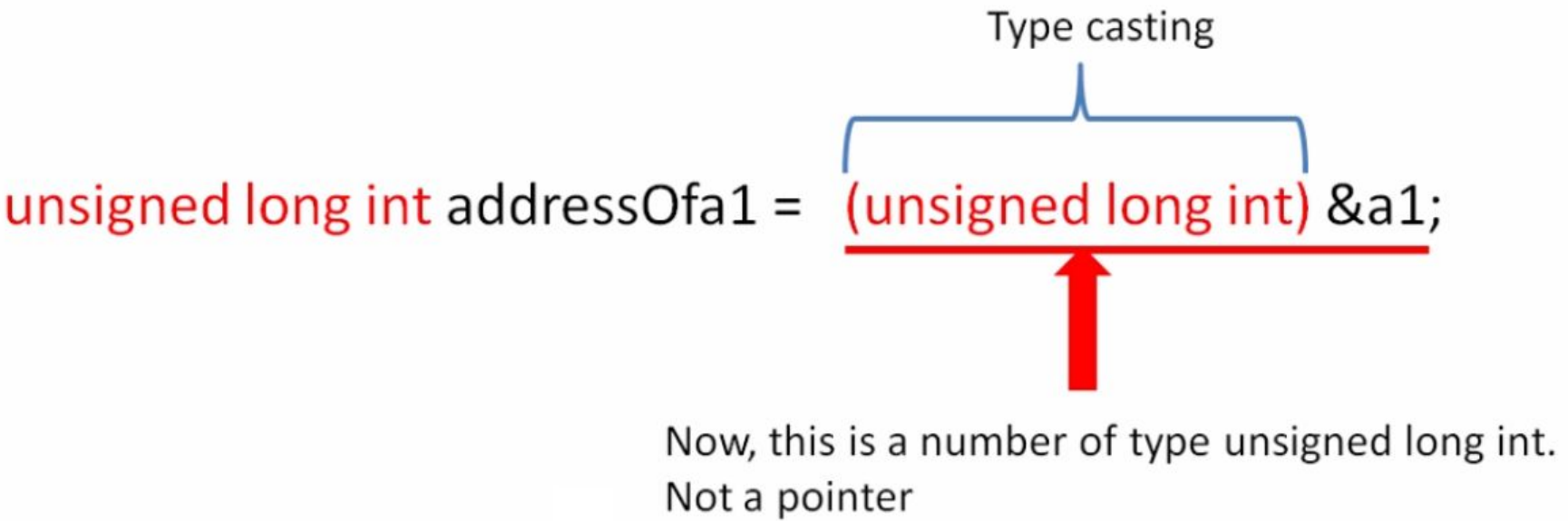
Write a program to print the address of variables.

```
int myData;
```


&myData → gives you memory location address of myData variable .


Memory location address. Address size is 8 bytes(64 bit machine )
A Memory location of computer memory
7 6 5 4 3 2 1 0
0 1 0 0 0 0 0 1
0x00007FFF8E3C3821
(a pointer)
a1
Variable name
Data stored in the memory location ( ASCII code of 'A')


This represent a variable.
Variables are represented by variable data type
This is a pointer data.
Pointer data are represented by pointer data type in 'C'
TYPE : unsigned long int
TYPE : char *
unsigned long int addressOfa1 = &a1;
We are trying to assign a pointer type data into a variable.
Hence there is a data type mismatch warning.
To solve this issue, convert pointer data type into variable data type using typecasting.
(more on this later)
This is not just a number.
This is a pointer


Type casting
unsigned long int addressOfa1 = (unsigned long int) &a1;
Now, this is a number of type unsigned long int.
Not a pointer

# Storage classes in ‘C’

## The Storage Classes in ‘C’ Language decides:

- ✓Scope of a variable
- ✓Visibility of a variable or function
- ✓Life time of a variable.

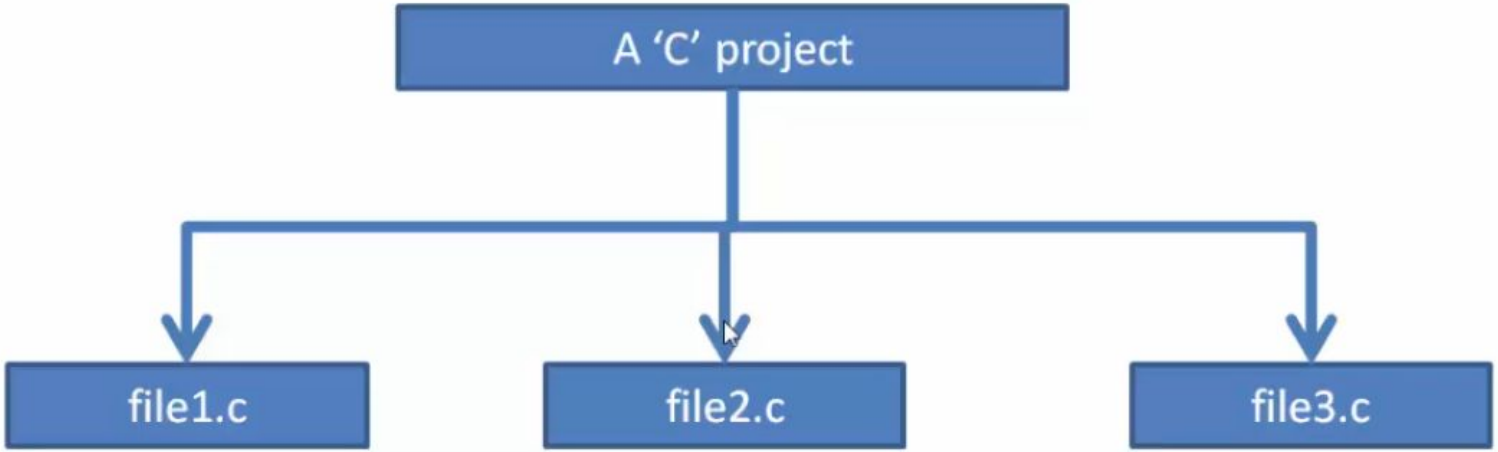
Scope, visibility, and lifetime are features of a variable . These features can be modified by using storage class specifiers.

# Storage class specifiers in ‘C’

**There are two widely used storage class specifiers in ‘C’**

- ✓ static
- ✓ extern

Now, our requirement for this program is, we want a global variable that is private to a function. We want a private variable that does not lose its existence even if the execution control goes out of the scope of that variable.




**A ‘C’ project can be a collection of multiple source files, and a ‘static’ storage class specifier can be used to manage the visibility of the variables across various files effectively.**

# Extern

'extern' storage class specifier is used to access the global variable , which is defined outside the scope of a file.

'extern' storage class specifier can also be used during the function call, when the function is defined outside the scope of the file.

# Extern

The keyword ‘extern’ is relevant only when your project consists of multiple files, and you need to access a variable defined in one file from another file.

'extern' keyword is used to extend the visibility of a function or variable.

# Functions in ‘C’

- In ‘C’, you write executable  statements inside a function .
- A ‘C.’ function is nothing but a collection of statements to perform a specific task.
- Every C program, at least, has one function called “main.”
- Using functions bring modularity to your code, easy to debug, modify and increases the maintainability of the code
- Using C function also minimizes code size and reducing code redundancy
- Functions provide abstraction. For example, printf is a function given by a standard library code. You need not worry about how it is implemented, and it serves the purpose.

- In your program, if you want to perform the same set of tasks again and again, then create a function to do that task and call it every time you need to perform that task.
- This reduces code redundancy and brings modularity to your program and also decreases the final executable size of your program.

# The general form of a function definition in C

```
return_data_type   function_name( parameter list )
{
    // body of the function
}
```

**main function definition with no input arguments.**

```
int main()
{
   /* ... */
}
```

**main function definition with 2 input arguments (Command line arguments ).**

```
int main(int argc, char* argv[] )
{
   /* ... */
}
```

**We don't use this definition of main for embedded systems because most of the time, there are no command-line arguments in embedded systems.**

# Function prototype (declaration )

- In ‘C’ functions first have to be declared before they are used.
- Prototype lets compiler to know about the return data type, argument list and their data type and order of arguments being passed to the function .

# Type casting

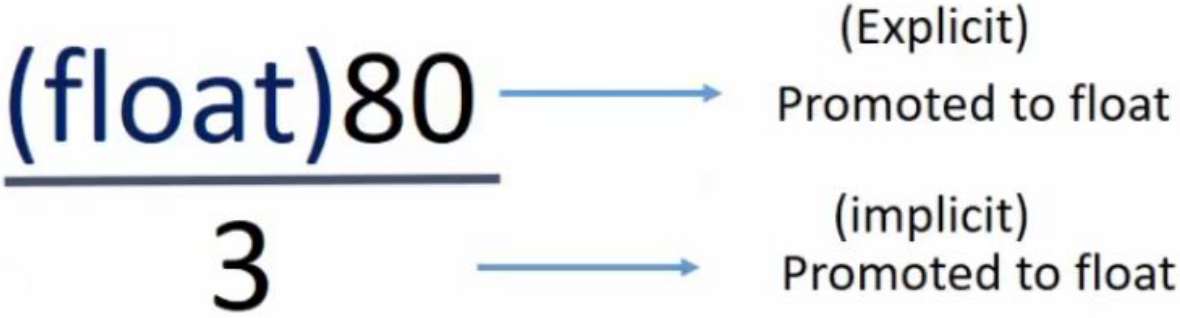
Typecasting is a way of converting a variable or data from one data type to another data type.

Data will be truncated when the higher data type is converted in to lower

# There are 2 types of casting

- Implicit casting  (Compiler does this )
- Explicit casting  ( Programmer does this )

# Explicit casting



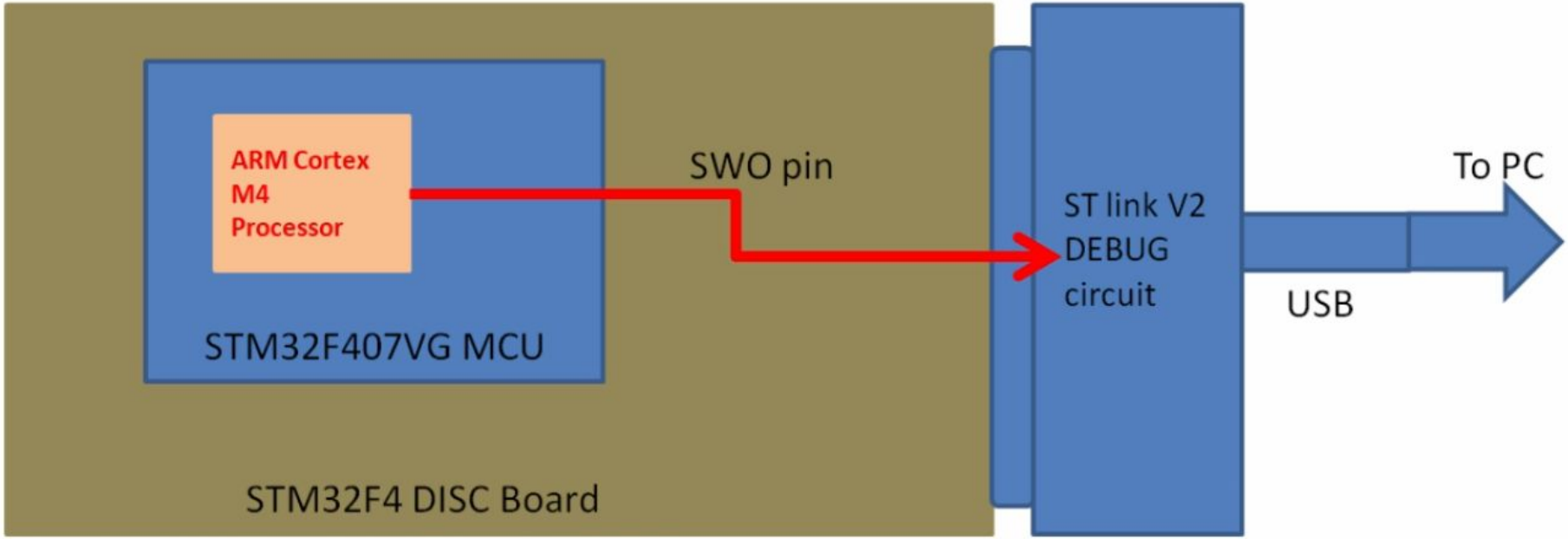

(So, the division is now float divided float which yields float)

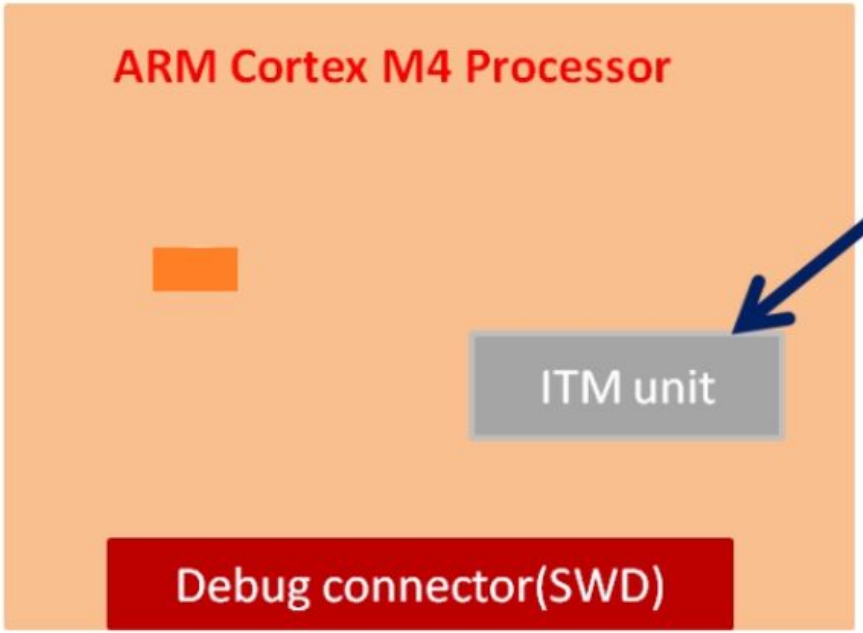
# Embedded – ‘Hello World’

How are we going to see the message output without a display device connected to our board?

# Using printf outputs on ARM Cortex M3/M4/M7 based MCUs

- This discussion is only applicable to MCUs based on ARM Cortex M3/M4/M7 or higher processors.
- printf works over SWO pin(Serial Wire Output) of SWD interface


ARM Cortex
M4
Processor
STM32F407VG MCU
STM32F4 DISC Board
SWO pin
ST link V2
DEBUG
circuit
USB
To PC




SWD(Serial Wire Debug)
2 pin (debug) + 1 pin (Trace)

Instrumentation Trace Macrocell Unit

**The ITM is an optional application-driven trace source that supports printf style debugging to trace operating system and application events, and generates diagnostic system information**

**Serial Wire Debug (SWD) is a two-wire protocol for accessing the ARM debug interface**

# SWD

✓Serial Wire Debug (SWD) is a two-wire protocol for accessing the ARM debug interface

✓It is part of the ARM Debug Interface Specification v5 and is an alternative to JTAG

✓The physical layer of SWD consists of two lines

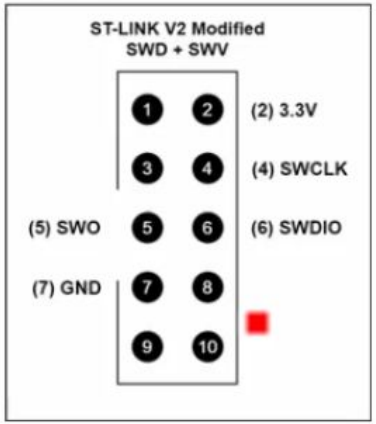
- •SWDIO: a bidirectional data line
- •SWCLK: a clock driven by the host

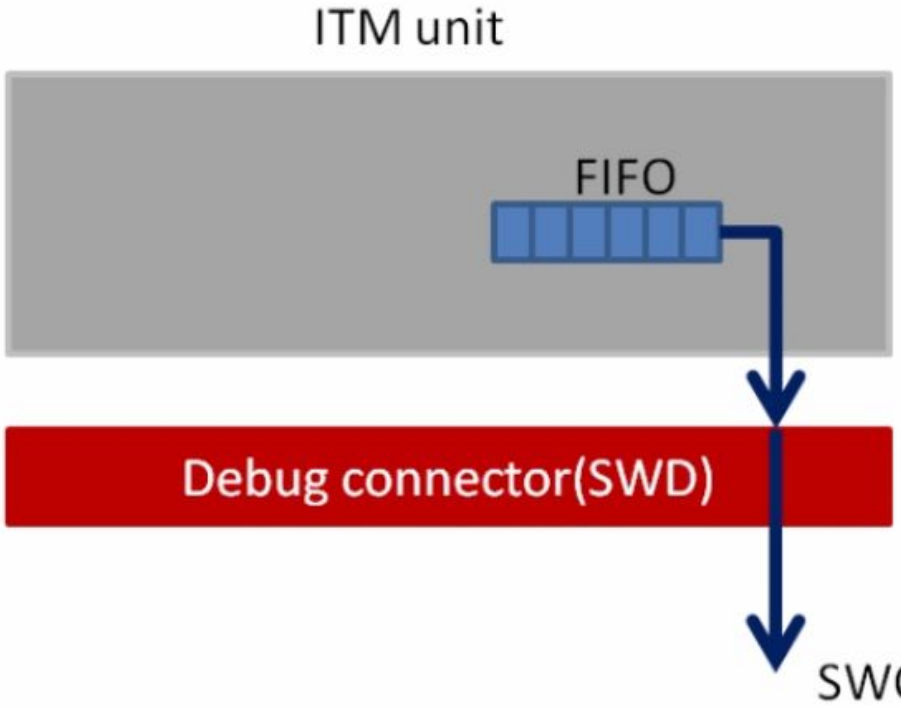
✓ By using SWD inteface should be able to program MCUs internal flash , you can access memory regions , add breakpoints, stop/run CPU.

✓The other good thing about SWD is you can use the serial wire viewer for your printf statements for debugging.

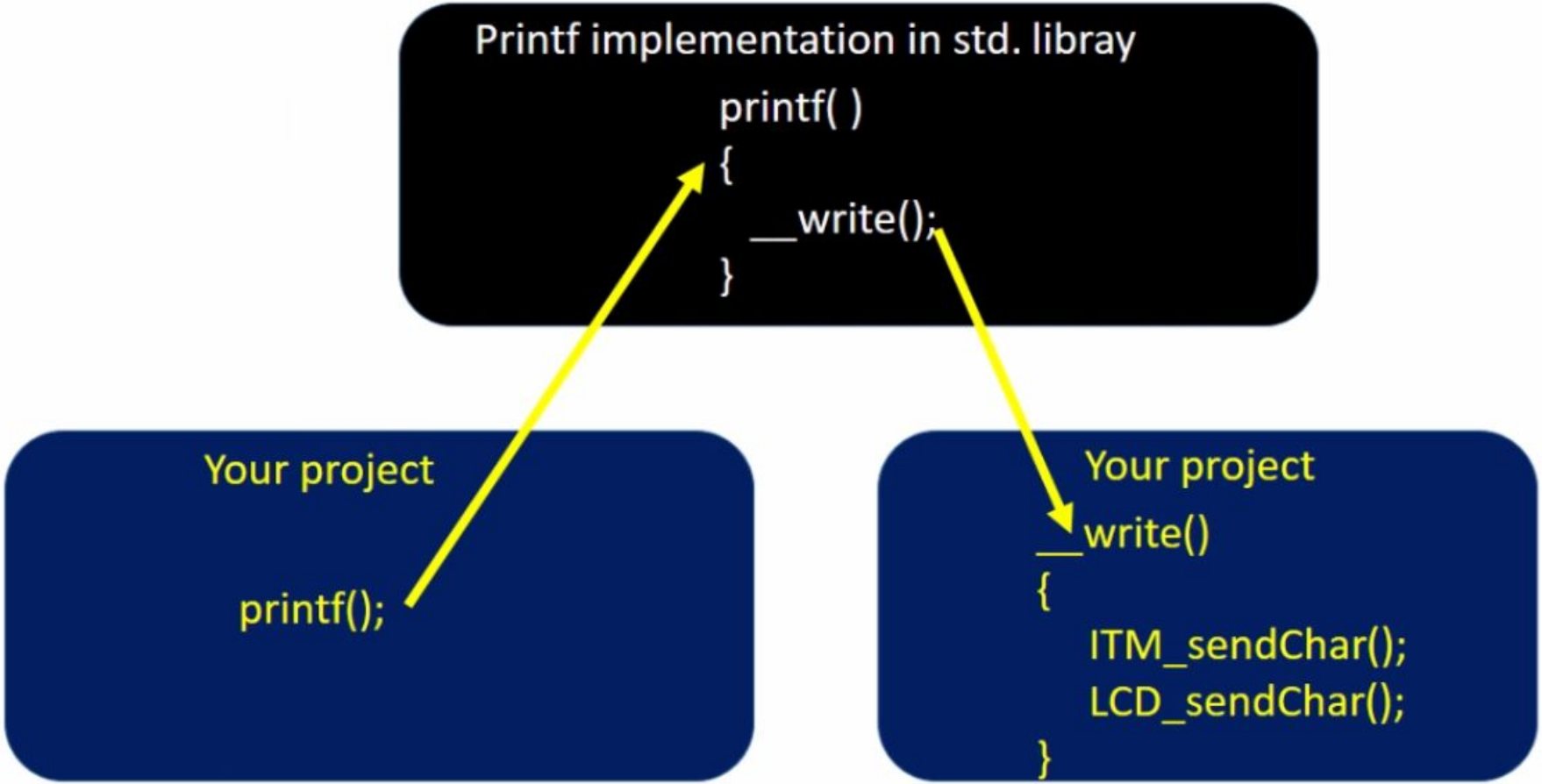
# SWD and JTAG

JTAG was the traditional mechanism for debug connections for ARM7/9 family, but with the Cortex-M family, ARM introduced the Serial Wire Debug (SWD) Interface. SWD is designed to reduce the pin count required for debug from the 4 used by JTAG (excluding GND) down to 2. In addition, SWD interface provides one more pin called SWO(Serial Wire Output) which is used for Single Wire Viewing (SWV), which is a low cost tracing technology






This SWO pin is connected to ST link circuitry of the board and can be captured using our debug software (IDE)


Printf implementation in std. libray
printf( )
{
__write();
}
Your project
printf();
Your project
__write()
{
ITM_sendChar();
LCD_sendChar();
}

# Cross compilation

X86/x86_64 architecture

Host Machine

ARM Architecture

Target machine

Produces executable for different architecture

Cross compiler
arm-none-eabi-gcc

.elf / .bin /.hex

This compiler runs on host machine

# Native compilation

Host Machine

Host native compiler
(mingw_64 or gcc)

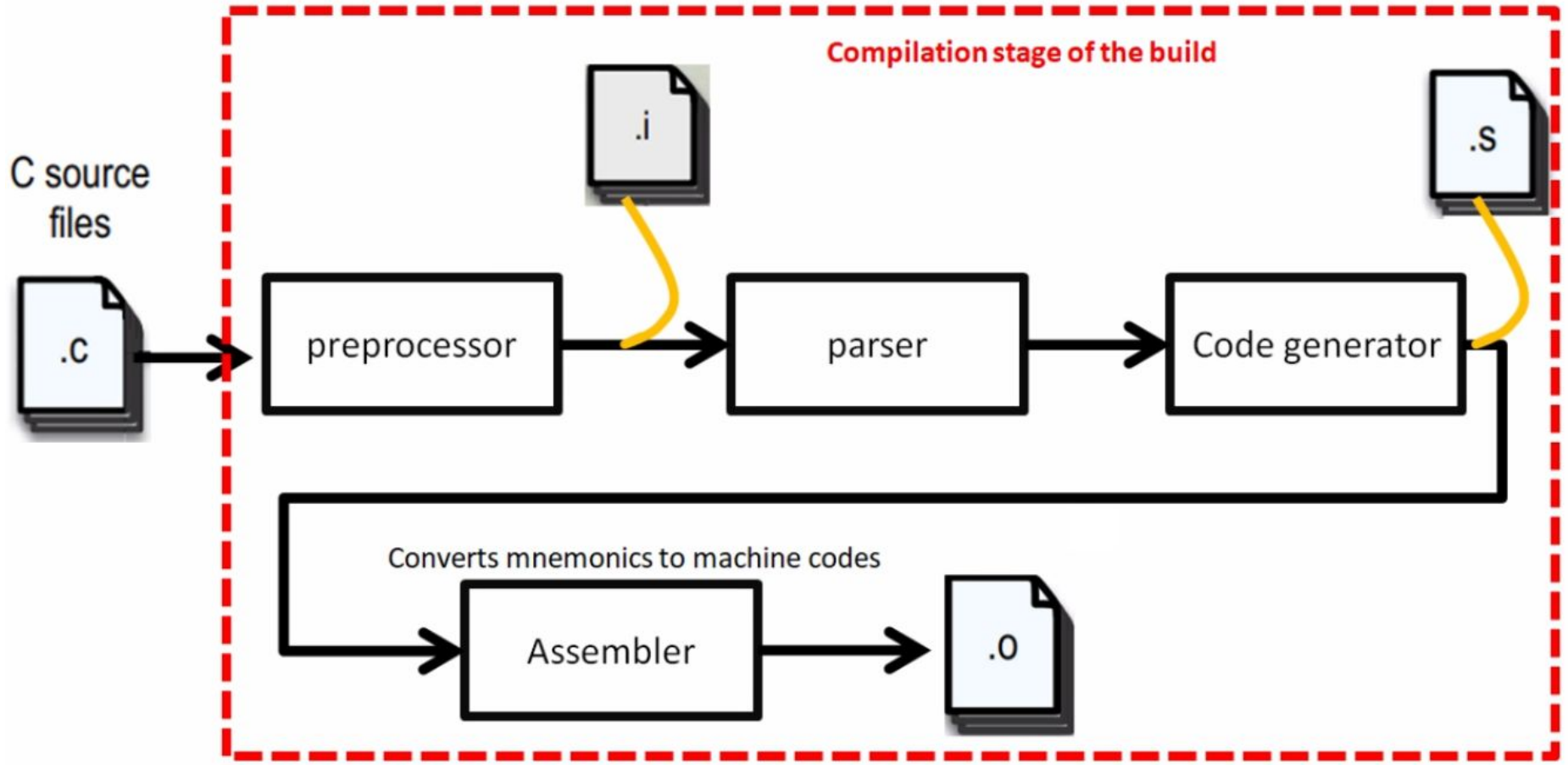
This compiler runs on host machine

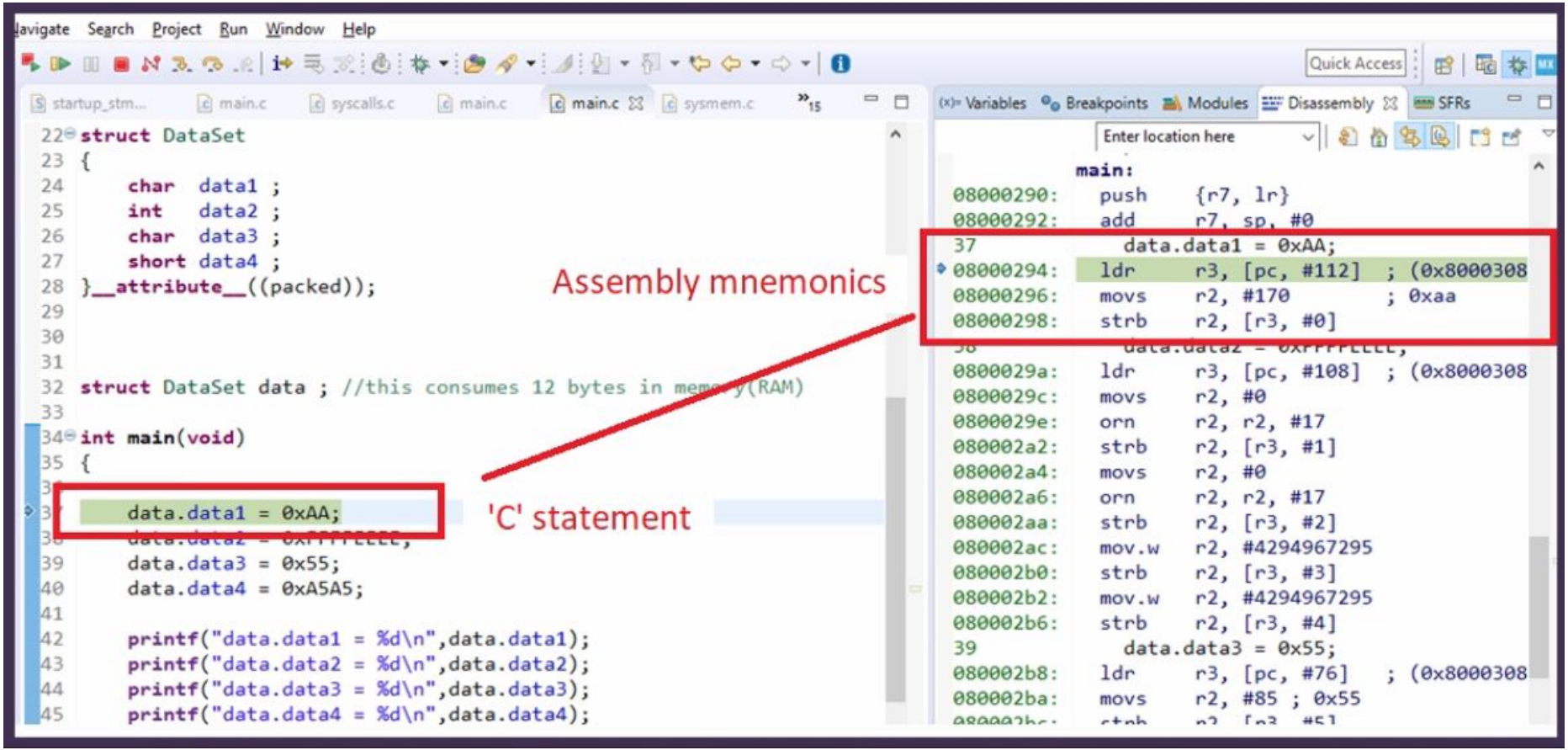
.exe

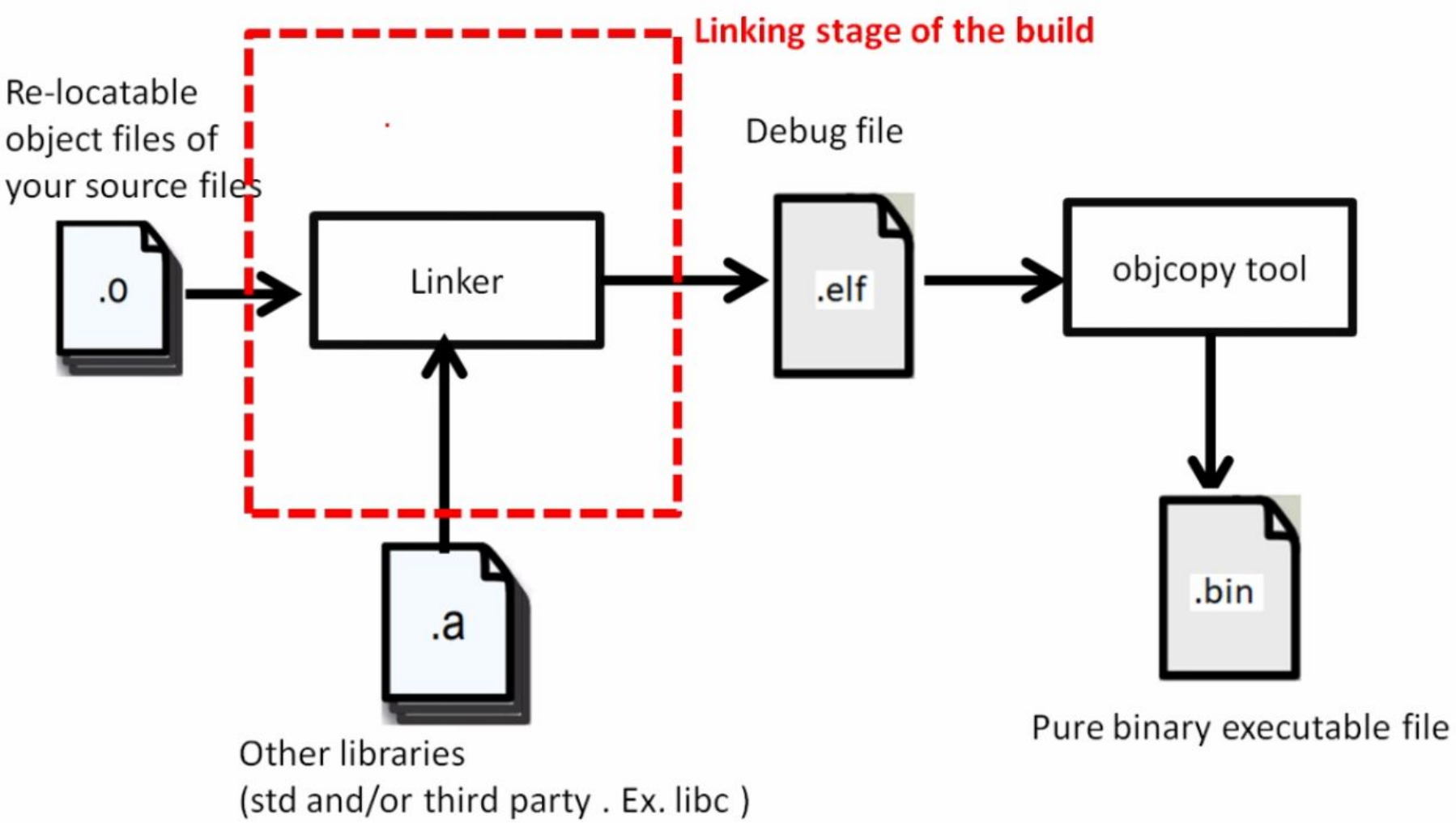
Produces executable which also runs on the same machine

# Build process

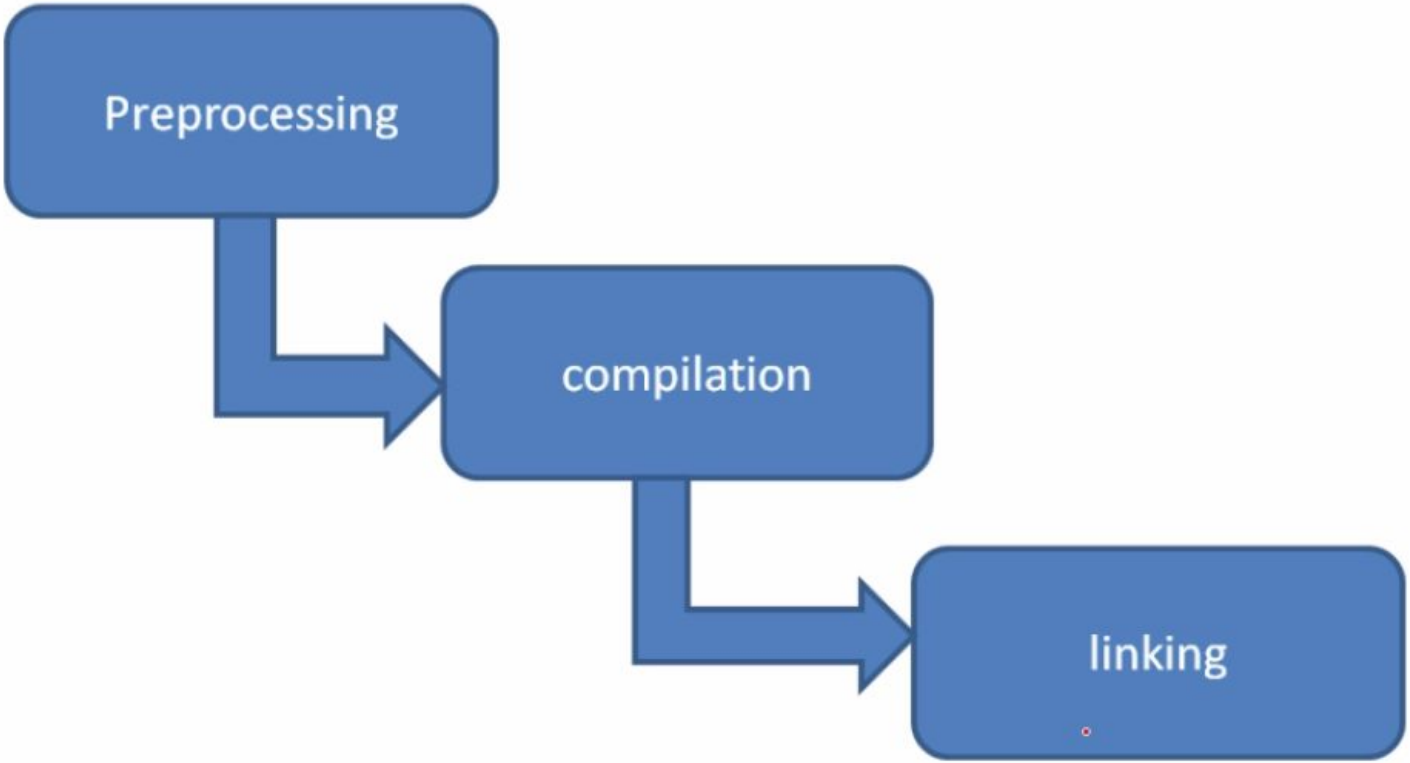
- Preprocessing
- Parsing
- Producing object file(s)
- Linking object files(s)
- Producing final executable
- Post processing of final executable


Compilation stage of the build
C source files
.c
preprocessor
.i
parser
Code generator
.s
Converts mnemonics to machine codes
Assembler
.o


Navigate Search Project Run Window Help
Quick Access
startup_stm... main.c syscalls.c main.c main.c sysmem.c
Variables Breakpoints Modules Disassembly SFRs
Enter location here
22 struct DataSet
23 {
24 char data1 ;
25 int data2 ;
26 char data3 ;
27 short data4 ;
28 }__attribute__((packed));
32 struct DataSet data ; //this consumes 12 bytes in memory(RAM)
34 int main(void)
35 {
37 data.data1 = 0xAA;
39 data.data3 = 0x55;
40 data.data4 = 0xA5A5;
42 printf("data.data1 = %d\n",data.data1);
43 printf("data.data2 = %d\n",data.data2);
44 printf("data.data3 = %d\n",data.data3);
45 printf("data.data4 = %d\n",data.data4);
Assembly mnemonics
'C' statement
main:
08000290: push {r7, lr}
08000292: add r7, sp, #0
37 data.data1 = 0xAA;
08000294: ldr r3, [pc, #112] ; (0x8000308
08000296: movs r2, #170 ; 0xaa
08000298: strb r2, [r3, #0]
0800029a: ldr r3, [pc, #108] ; (0x8000308
0800029c: movs r2, #0
0800029e: orn r2, r2, #17
080002a2: strb r2, [r3, #1]
080002a4: movs r2, #0
080002a6: orn r2, r2, #17
080002aa: strb r2, [r3, #2]
080002ac: mov.w r2, #4294967295
080002b0: strb r2, [r3, #3]
080002b2: mov.w r2, #4294967295
080002b6: strb r2, [r3, #4]
39 data.data3 = 0x55;
080002b8: ldr r3, [pc, #76] ; (0x8000308
080002ba: movs r2, #85 ; 0x55


Linking stage of the build
Re-locatable
object files of
your source files
.o
Linker
.a
Other libraries
(std and/or third party . Ex. libc )
Debug file
.elf
objcopy tool
.bin
Pure binary executable file

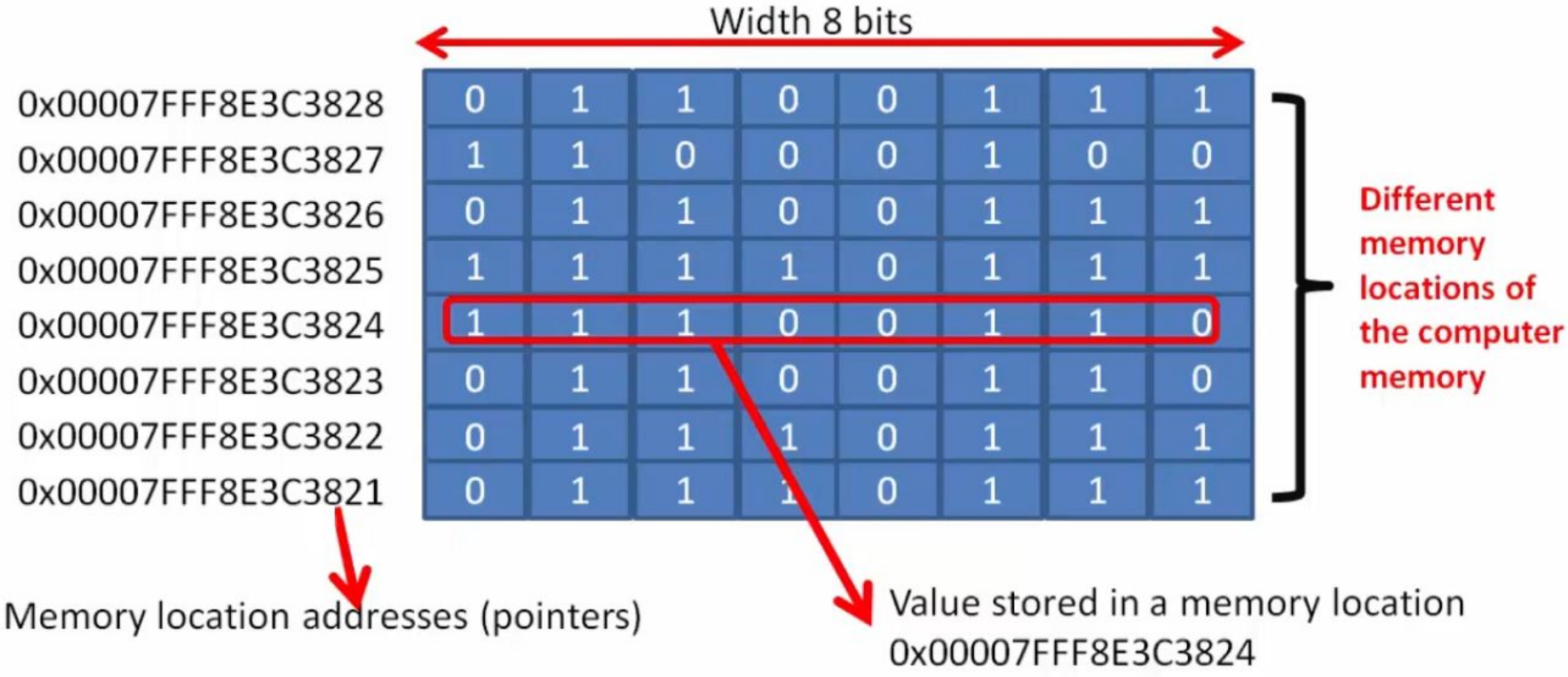
# Summary of build process



# Pointers in ‘C’

# Pointers in ‘C’

- Pointers are one of the essential programming features which are available in ‘C.’
- Pointers make ‘C’ programming more powerful
- Pointers are heavily used in embedded `C` programming to
  - Configure the peripheral register addresses
  - Read/Write into peripheral data registers
  - Read/Write into SRAM/FLASH locations and for many other things.




***On 64 bit machine , the pointer size ( memory location address size) is 8 bytes (64bit)***

Now, let's see how to store a pointer inside the program.

# Pointer variable definition

<pointer data type > <Variable Name > ;

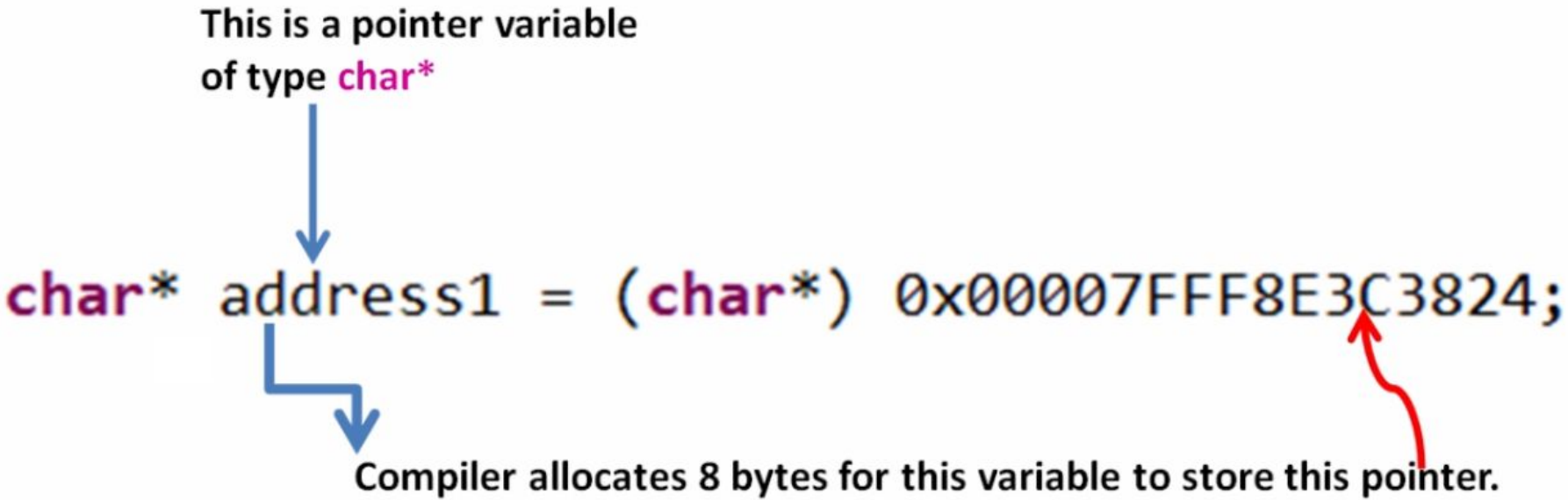
# Pointer data types

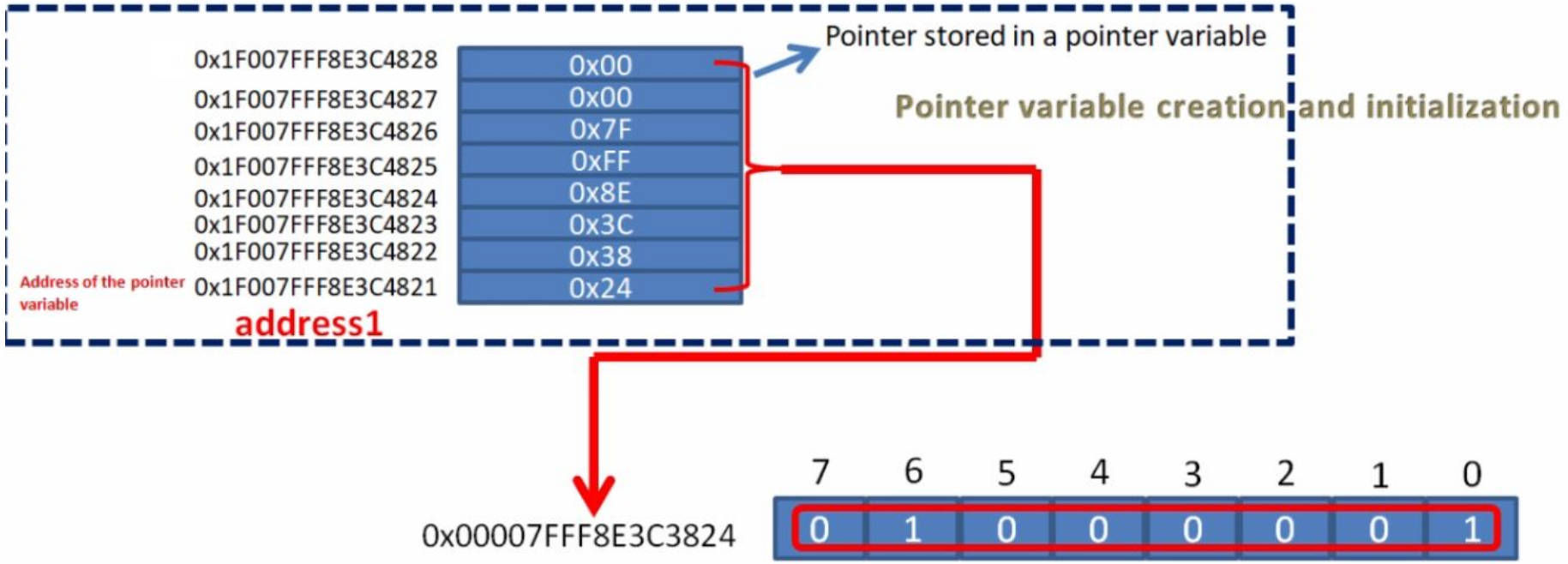
During variable declaration asterisk(*) is used to differentiate between a pointer variable declaration and a non pointer variable declaration.

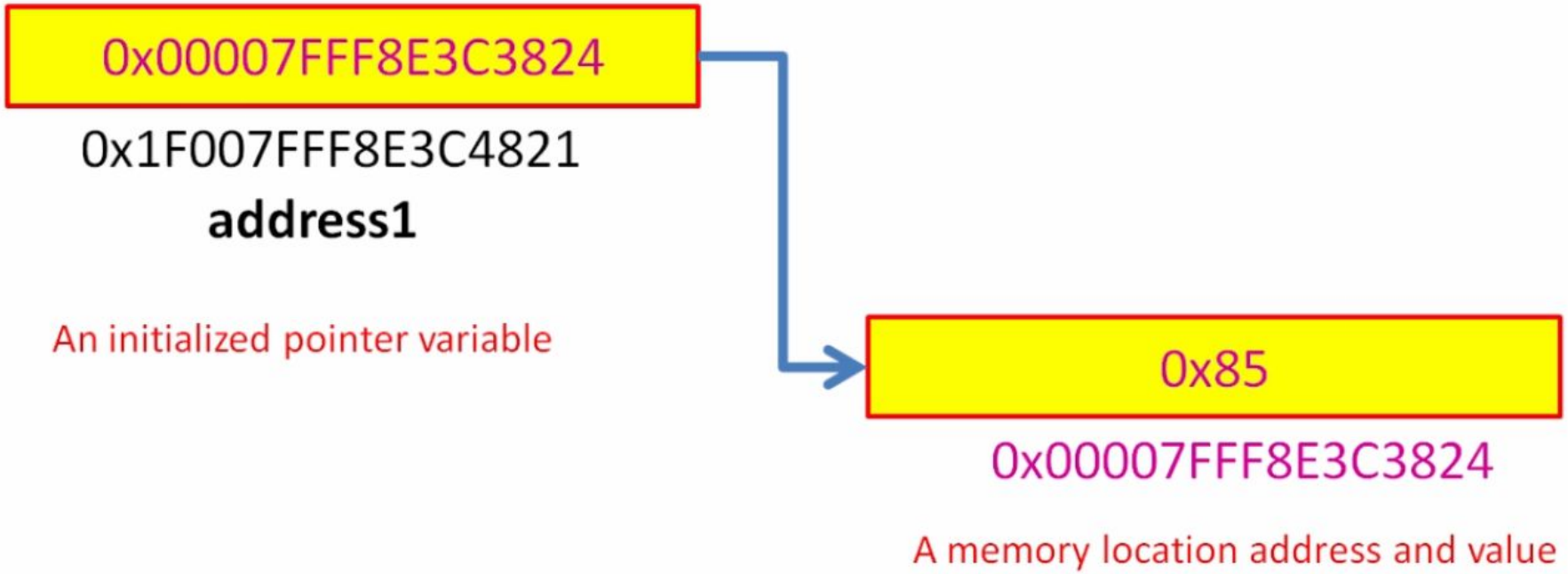
char*
int*
long long int*
float*
double*

unsigned char*
unsigned int*
unsigned long long int*
unsigned float*
unsigned double*

# Pointer variable definition and initialization




Pointer stored in a pointer variable
Pointer variable creation and initialization
0x1F007FFF8E3C4828
0x1F007FFF8E3C4827
0x1F007FFF8E3C4826
0x1F007FFF8E3C4825
0x1F007FFF8E3C4824
0x1F007FFF8E3C4823
0x1F007FFF8E3C4822
0x1F007FFF8E3C4821
0x00
0x00
0x7F
0xFF
0x8E
0x3C
0x38
0x24
Address of the pointer variable
address1
0x00007FFF8E3C3824
7 6 5 4 3 2 1 0
0 1 0 0 0 0 0 1


0x00007FFF8E3C3824
0x1F007FFF8E3C4821
address1
An initialized pointer variable
0x85
0x00007FFF8E3C3824
A memory location address and value

```
char* address1
int*  address1
long long int* address1
double* address1
```

The compiler will always reserve **8 bytes** for the pointer variable irrespective of their pointer data type.

In other words, the pointer data type doesn't control the memory size of the pointer variable.

# Pointer variable definition and initialization

Then, what is the purpose of mentioning “Pointer data type “?

?

```
char* address1 = (char*) 0x00007FFF8E3C3824;
```

?

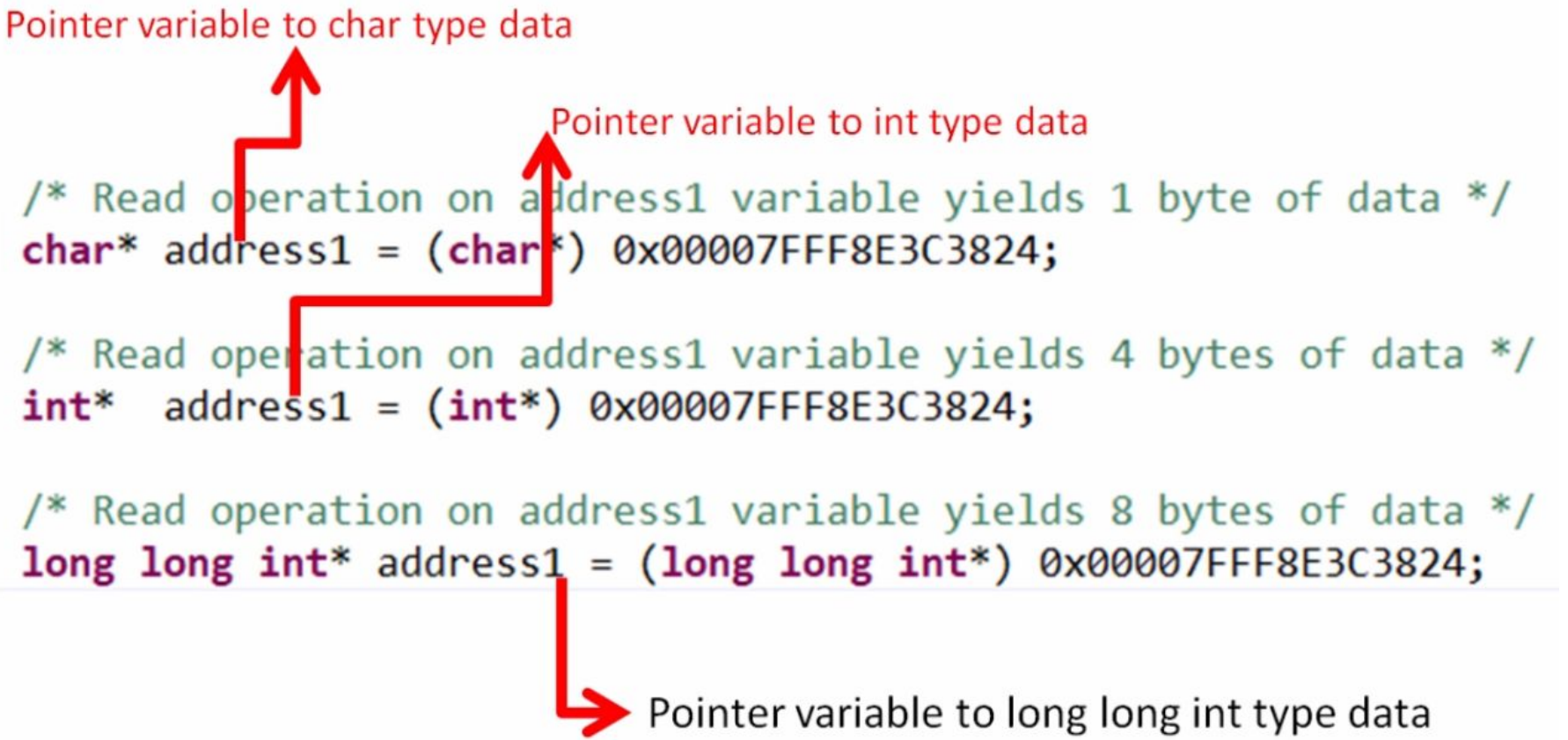
```
char* address1 = (char*) 0x00007FFF8E3C3824;
```

*The pointer data type decides the behavior of the operations carried out on the pointer variable.*
*Operations : read, write, increment, decrement*

```c
/* Read operation on address1 variable yields 1 byte of data */
char* address1 = (char*) 0x00007FFF8E3C3824;

/* Read operation on address1 variable yields 4 bytes of data */
int*  address1 = (int*) 0x00007FFF8E3C3824;

/* Read operation on address1 variable yields 8 bytes of data */
long long int* address1 = (long long int*) 0x00007FFF8E3C3824;
```


Pointer variable to char type data
Pointer variable to int type data
/* Read operation on address1 variable yields 1 byte of data */
char* address1 = (char*) 0x00007FFF8E3C3824;
/* Read operation on address1 variable yields 4 bytes of data */
int* address1 = (int*) 0x00007FFF8E3C3824;
/* Read operation on address1 variable yields 8 bytes of data */
long long int* address1 = (long long int*) 0x00007FFF8E3C3824;
Pointer variable to long long int type data

# Read operation on the pointer

```
char* address1 = (char*) 0x00007FFF8E3C3824;
```

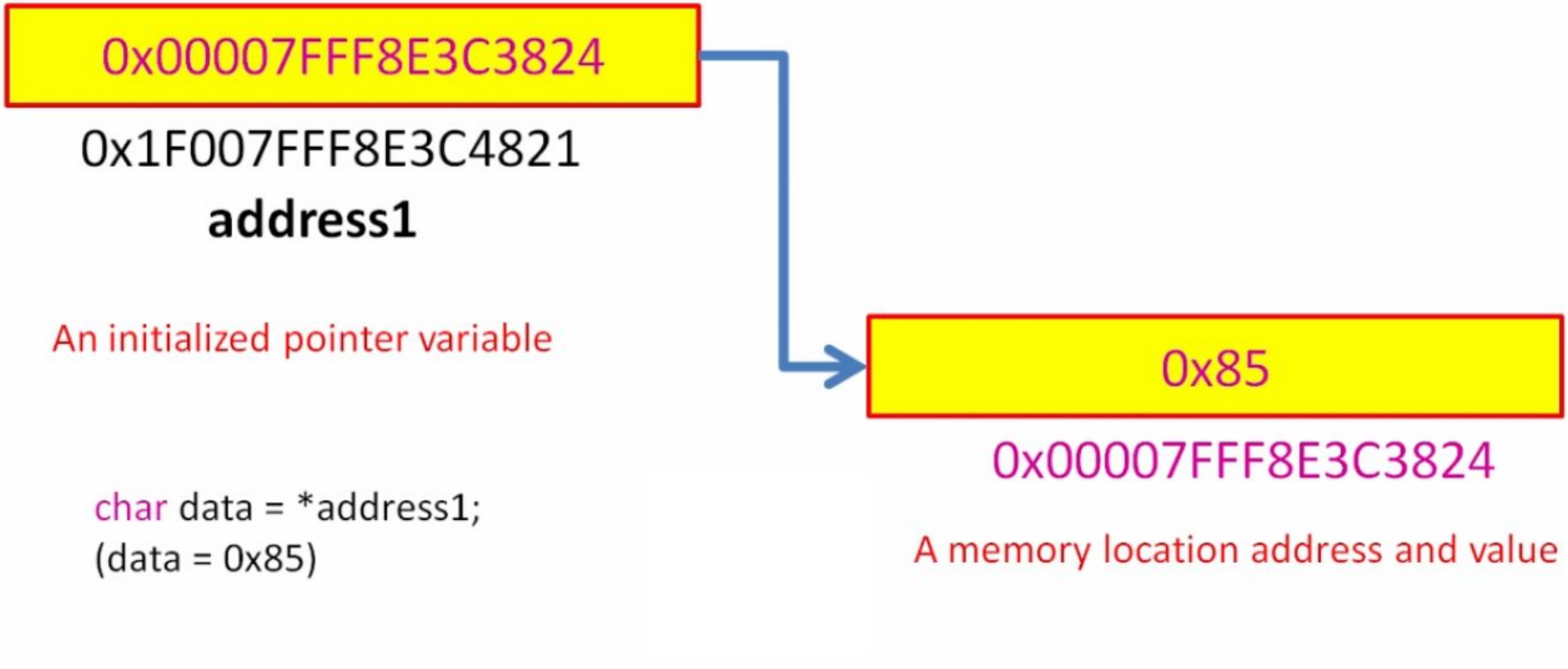
## Read data from the pointer

char data = *address1; //dereferencing a pointer to read data

*1 byte of data is read from the pointer and stored in to "data" variable*

**** : "Value at Address" Operator***

***& : " Address of" operator***

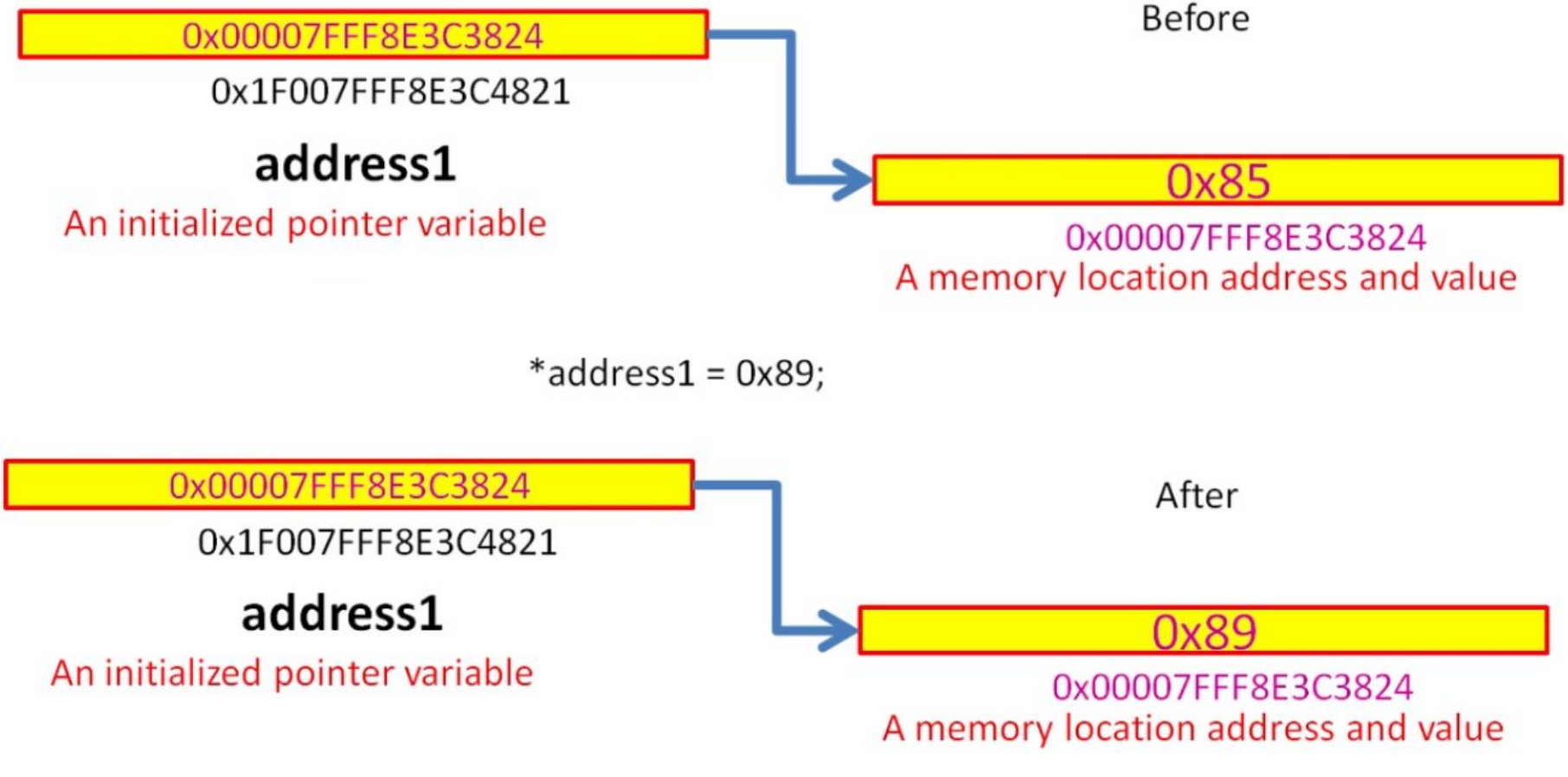

0x00007FFF8E3C3824
0x1F007FFF8E3C4821
address1
An initialized pointer variable
char data = *address1;
(data = 0x85)
0x85
0x00007FFF8E3C3824
A memory location address and value

# Write operation on the pointer

```
char* address1 = (char*) 0x00007FFF8E3C3824;
```

Write data to a pointer

*address1 = 0x89; //dereferencing a pointer to write data


0x00007FFF8E3C3824
0x1F007FFF8E3C4821
address1
An initialized pointer variable
Before
0x85
0x00007FFF8E3C3824
A memory location address and value
*address1 = 0x89;
0x00007FFF8E3C3824
0x1F007FFF8E3C4821
address1
An initialized pointer variable
After
0x89
0x00007FFF8E3C3824
A memory location address and value

# Exercise

1) Create a char type variable and initialize it to value 100
2) Print the address of the above variable.
3) Create a pointer variable and store the address of the above variable
4) Perform read operation on the pointer variable to fetch 1 byte of data from the pointer
5) Print the data obtained from the read operation on the pointer.
6) Perform write operation on the pointer to store the value 65
7) Print the value of the variable defined in step 1

# Bitwise Operators in ‘C’

# Bitwise Operators in ‘C’

| | |
|---|---|
| & (bitwise AND) | >> (bitwise Right Shift) |
| \| (bitwise OR) | Unary ~ (bitwise NOT) (Negation) |
| << (bitwise Left Shift) | ^ (bitwise XOR) |

# Difference between Logical operator and bitwise operator

- && is a 'logical AND' operator
- & is a 'bitwise AND' operator

```
char A = 40;
char B = 30;
```

C = A && B; ?

C = A & B; ?

```
char A = 40;    char B = 30;  char C;
```

| C = A && B; | C = A & B; |
| --- | --- |
| (Logical) | (Bitwise) |

C=1;

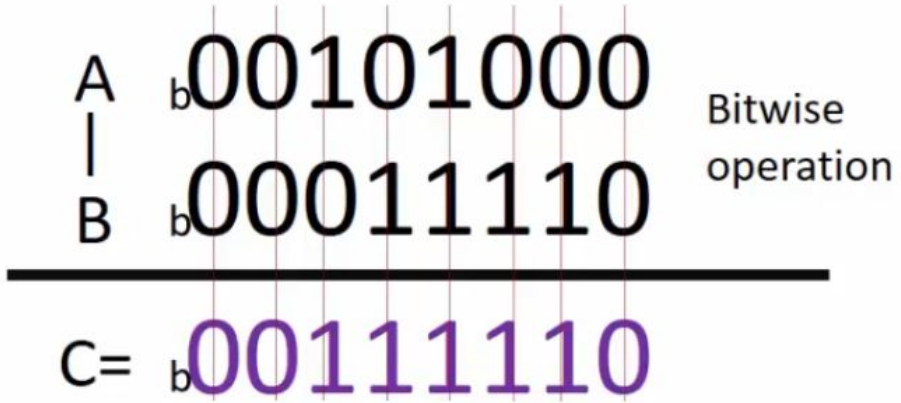
```
A  b00101000
&
B  b00011110
-----------
C= b00001000
```

Bitwise operation

```
char A = 40;    char B = 30;  char C;
```

```
C = A || B;                                C = A | B;
```

C=1;



```
char A = 40;    char B = 30;  char C;
```

| C = ! A; | C = ~A; |
|---|---|

C=0;

A $_b$00101000

C= ~A $_b$11010111

Bitwise operation

C= -41

```
char A = 40;    char B = 30;  char C;
```

```
C = A ^ B;
```

**Truth table for XOR Gate**

| INPUTS | | OUTPUTS |
|---|---|---|
| A | B | Y = A ^ B |
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |

```
A  b00101000
^
B  b00011110
-----------
C= b00110110
```

Bitwise operation

C=54

# Applicability of bitwise operations

In an Embedded C program, most of the time you will be doing,

- ✓Testing of bits ( &)
- ✓Setting of bits (|)
- ✓Clearing of bits (~ and &)
- ✓Toggling of bits (^)

# Exercise

- Write a program which takes 2 integers from the user , computes bitwise &,|,^ and ~ and prints the result.

# Applicability of bitwise operations

In an Embedded C program, most of the time you will be doing,

- ✓Testing of bits ( &)
- ✓Setting of bits (|)
- ✓Clearing of bits (~ and &)
- ✓Toggling of bits (^)

# Exercise: Testing of bits

- Write a program to find out whether a user entered integer is even or odd.
- Print an appropriate message on the console.
- Use testing of bits logic

46 (0x2E)

lsb

b00101110

Number is EVEN

47 (0x2F)

lsb

b00101111

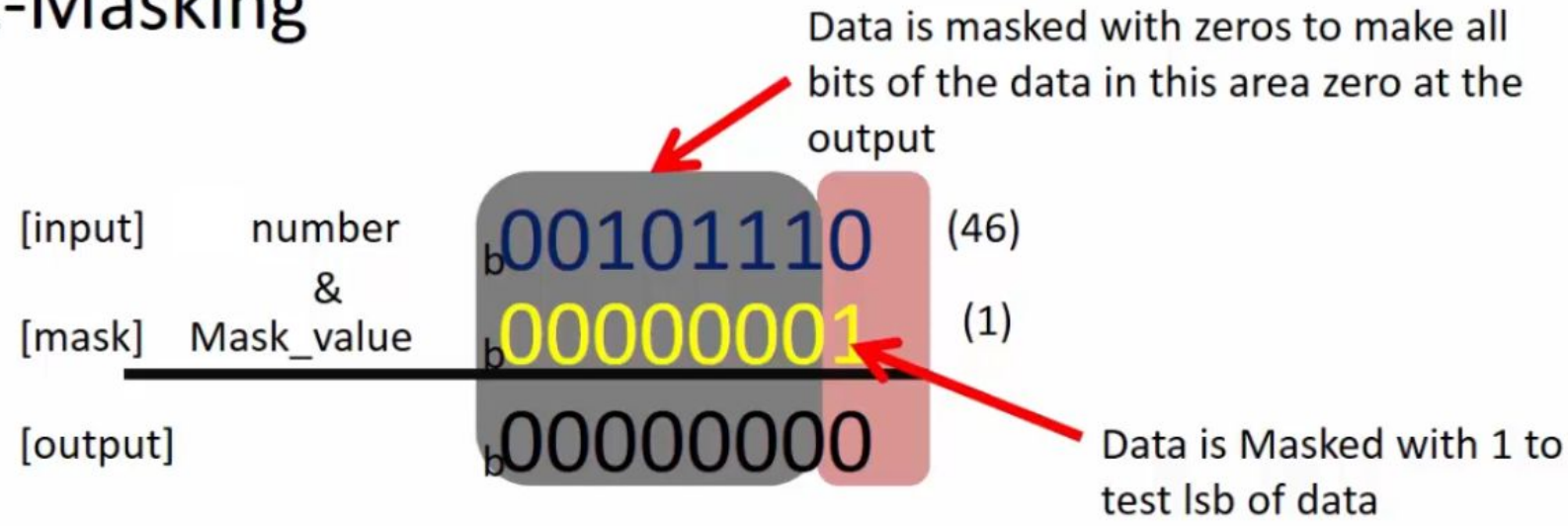
Number is ODD

Test the least significant bit of the number using bitwise operation

If the lsb is zero, then number is EVEN

If the lsb is one, then number is ODD

# Bit-Masking




Bit Masking is a technique in programming used to test or modify the states of the bits of a given data.

Modify : if the state of the bit is zero, make it one or if the state of the bit is 1 then make it 0
Test : check whether the required bit position of a data is 0 or 1

7 6 5 4 3 2 1 0

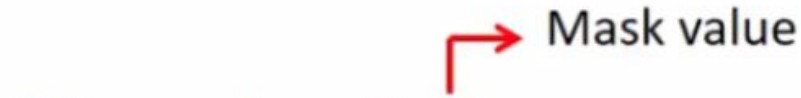
| | |
|---|---|
| [data] | 00101110 |
| & [Mask] | 00000010 |
| [output] | 00000010 |

| | |
|---|---|
| [data] | 10101110 |
| & [Mask] | 10000000 |
| [output] | 10000000 |

| | |
|---|---|
| [data] | 00101110 |
| & [Mask] | 00000011 |
| [output] | 00000010 |

| | |
|---|---|
| [data] | 00101110 |
| & [Mask] | 00010000 |
| [output] | 00000000 |

# EVEN or ODD using Bitwise operation

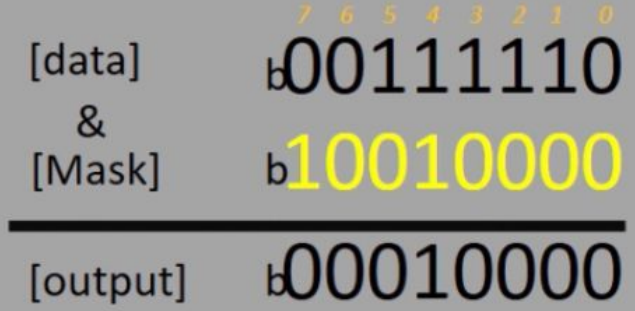



```
If ( number & 1)
{
        print(number odd)
}else
{
        print(Number even)
}
```

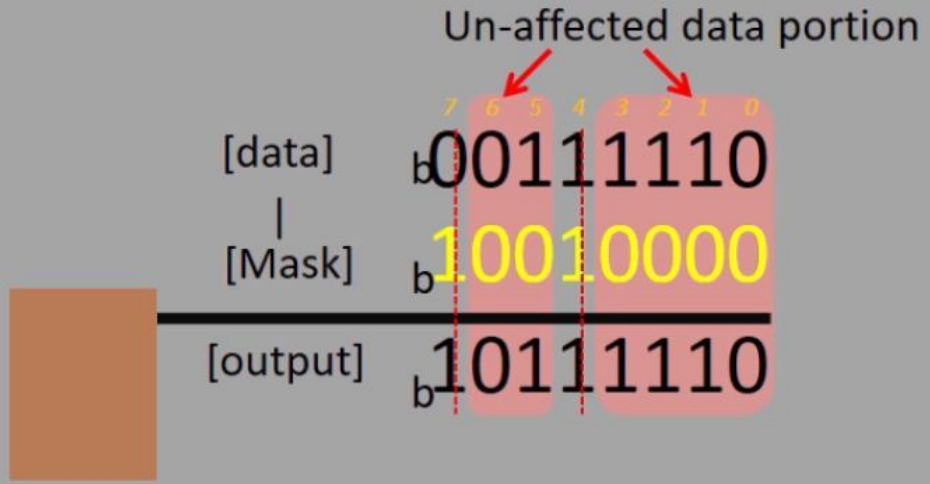
# Exercise: Setting of bits

- Write a program to set(make bit state to 1) 4th and 7th-bit position of a given number and print the result.

# Exercise: Setting of bits

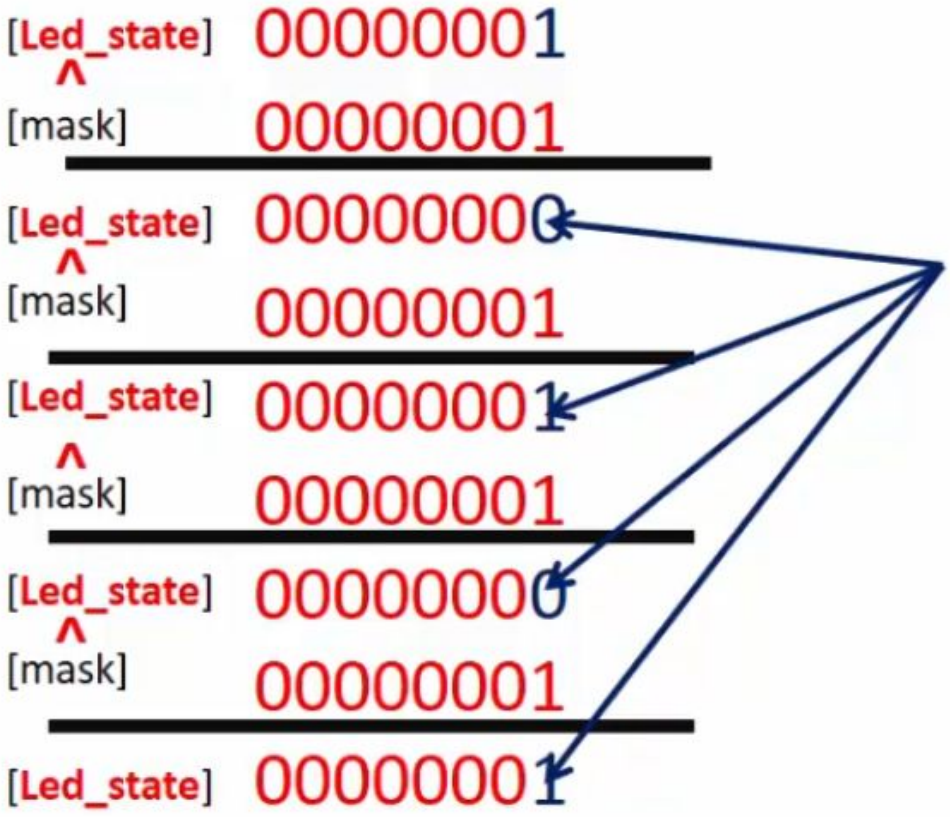



*'&' is used to 'TEST' not to 'SET'*




*'|' is used to 'SET' not to 'TEST'*

# Toggling of bits




Led_state = Led_state ^ 0x01

Output toggles

# Applicability of bitwise operations in Embedded Systems

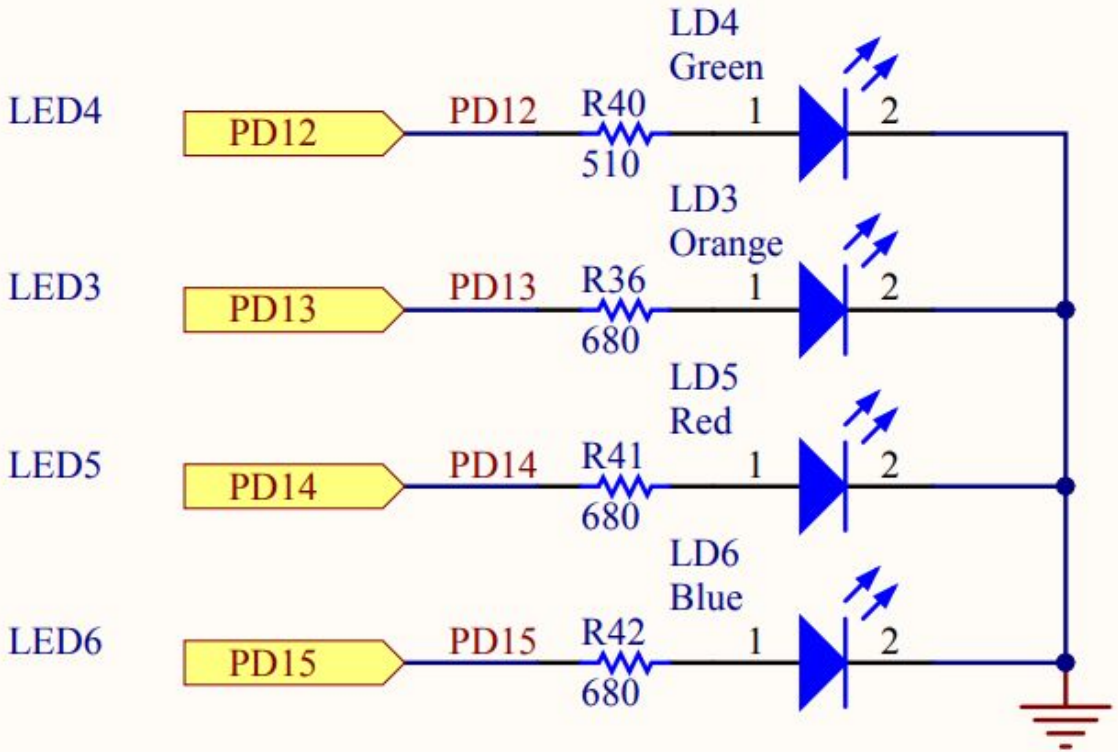
# Exercise

- Write a program to turn on the LED of your target board
- For this exercise we need the knowledge of
  - Pointers
  - Bitwise operations
  - Hardware connections

# Hardware connections

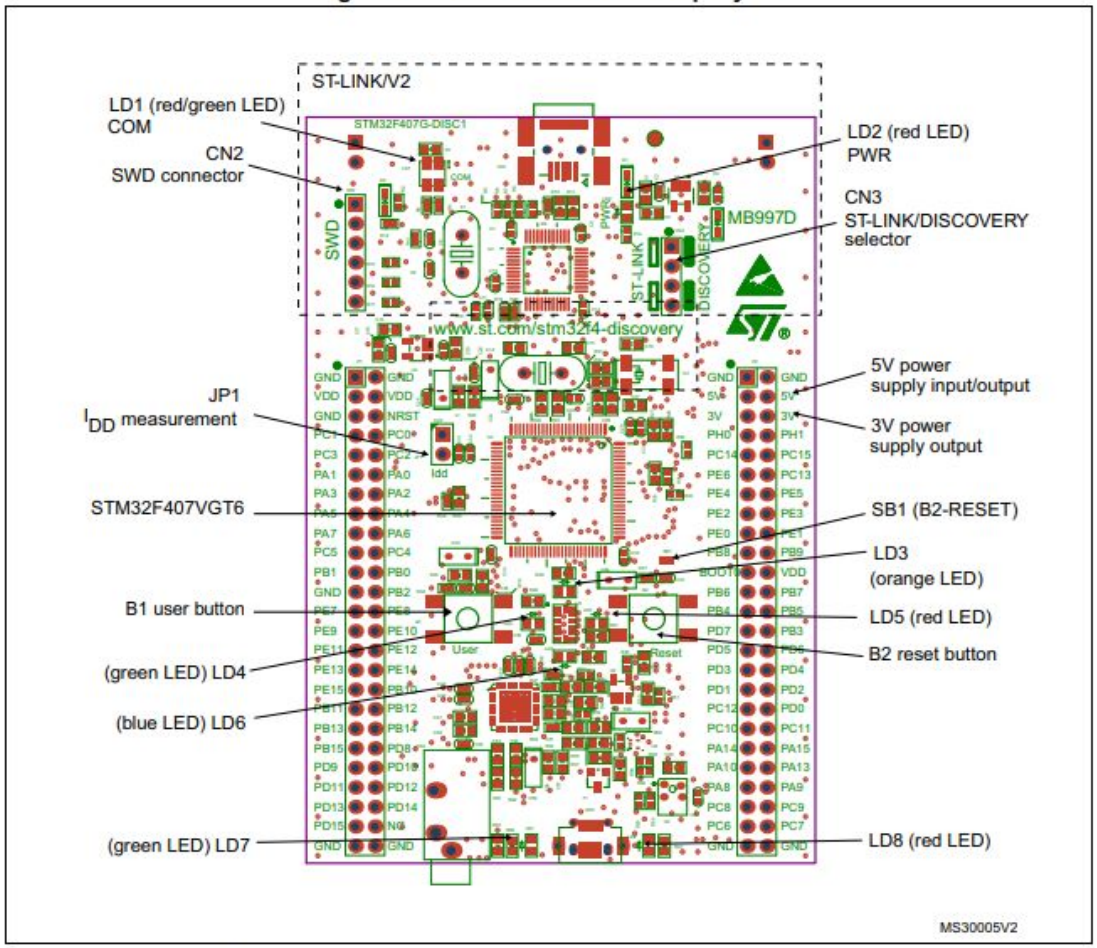
- Lets understand how external hardware (LED) is connected to MCU
  - Refer schematic of the board you are using

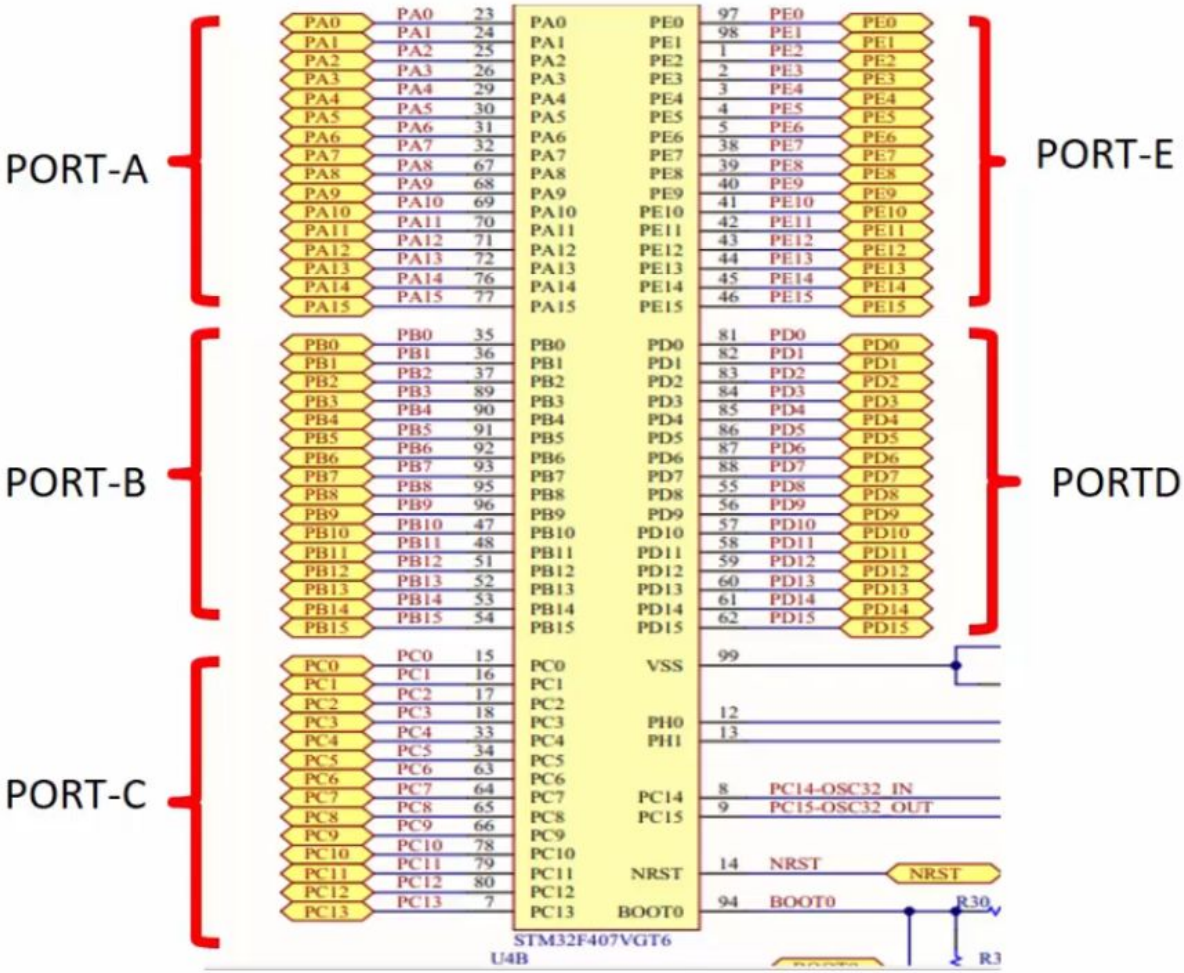
https://www.st.com/en/evaluation-tools/stm32f4discovery.html#cad-resources


LED4
PD12
PD12
R40
510
LD4
Green
1
2
LED3
PD13
PD13
R36
680
LD3
Orange
1
2
LED5
PD14
PD14
R41
680
LD5
Red
1
2
LED6
PD15
PD15
R42
680
LD6
Blue
1
2

# LEDs

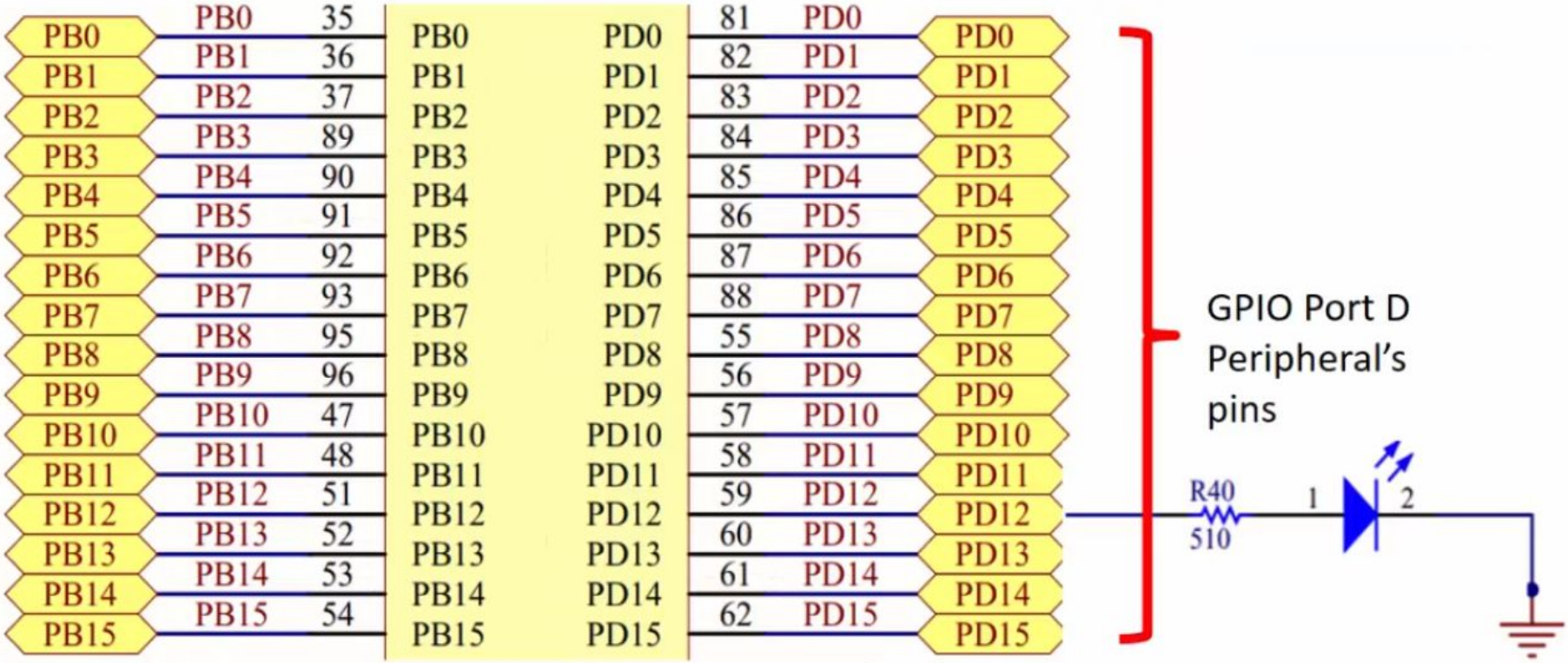
**Figure 3. STM32F4DISCOVERY top layout**






In STM32Fx based MCUs , each port has 16 pins where you can connect external peripherals .
(LED, Display , button, Bluetooth transceiver , external memory (e.g. EEPROM), Joy stick , keypad ,etc )

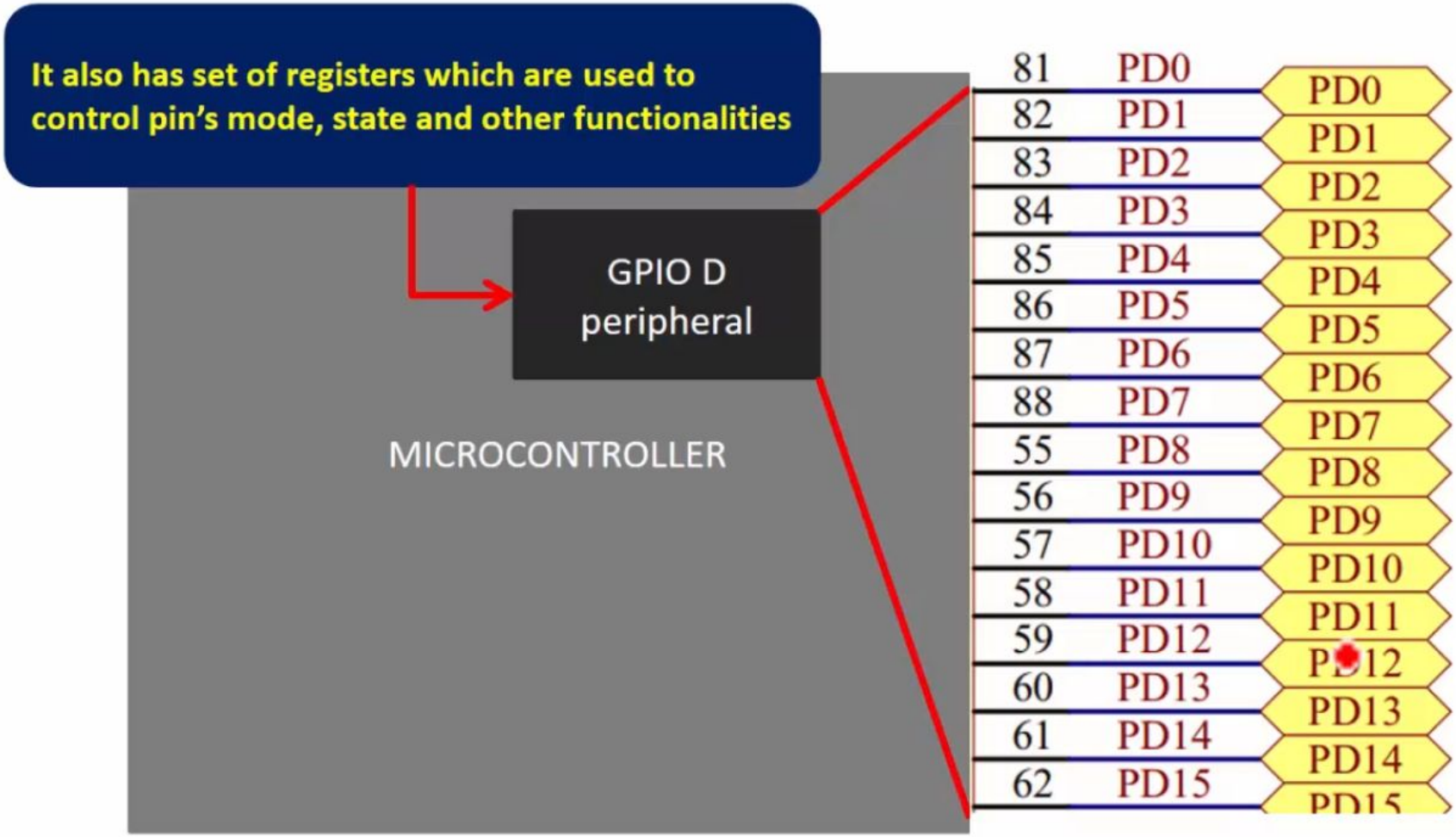
How do you access 12th pin of GPIO PORT-D peripheral from software ?



Now your goal is to control the I/O pin PD12's state either HIGH or LOW through software to make LED turn ON or OFF.

PD12 : 12$^{th}$ pin of the GPIO PORT D peripheral

**GPIO: General Purpose Input Output*


It also has set of registers which are used to control pin's mode, state and other functionalities
GPIO D peripheral
MICROCONTROLLER
81 PD0
82 PD1
83 PD2
84 PD3
85 PD4
86 PD5
87 PD6
88 PD7
55 PD8
56 PD9
57 PD10
58 PD11
59 PD12
60 PD13
61 PD14
62 PD15
PD0
PD1
PD2
PD3
PD4
PD5
PD6
PD7
PD8
PD9
PD10
PD11
PD12
PD13
PD14
PD15

# Bitwise Operators in ‘C’


(bitwise AND)


(bitwise Right Shift)

Unary


(bitwise OR)


(bitwise NOT) (Negation)


(bitwise Left Shift)


(bitwise XOR)

# Bitwise right shift operator(>>)

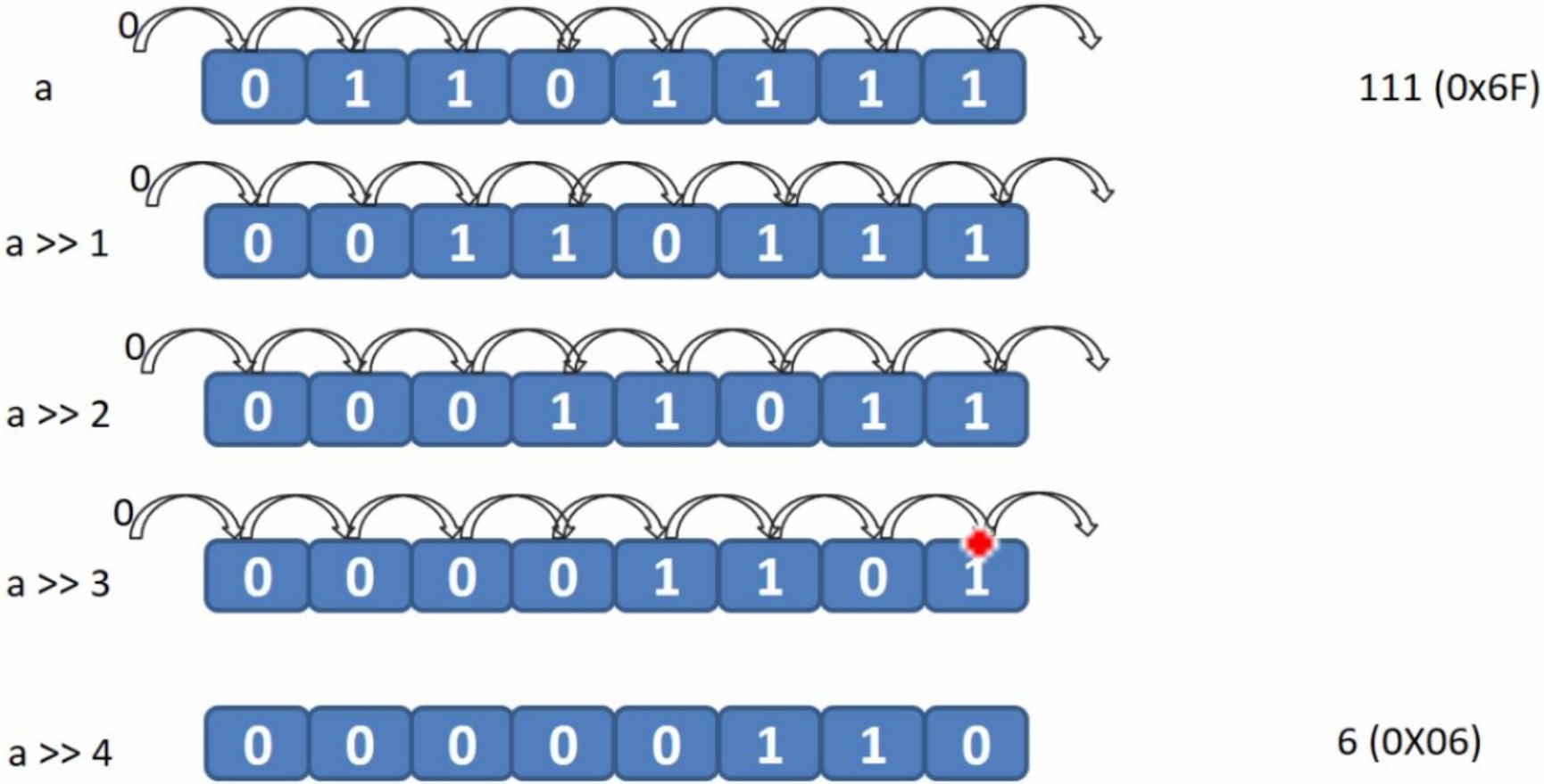
- This operator takes 2 operands
- Bits of the 1st operand will be right shifted by the amount decided by the 2nd operand
- Syntax : operand1 >> operand2
- Lets see an example

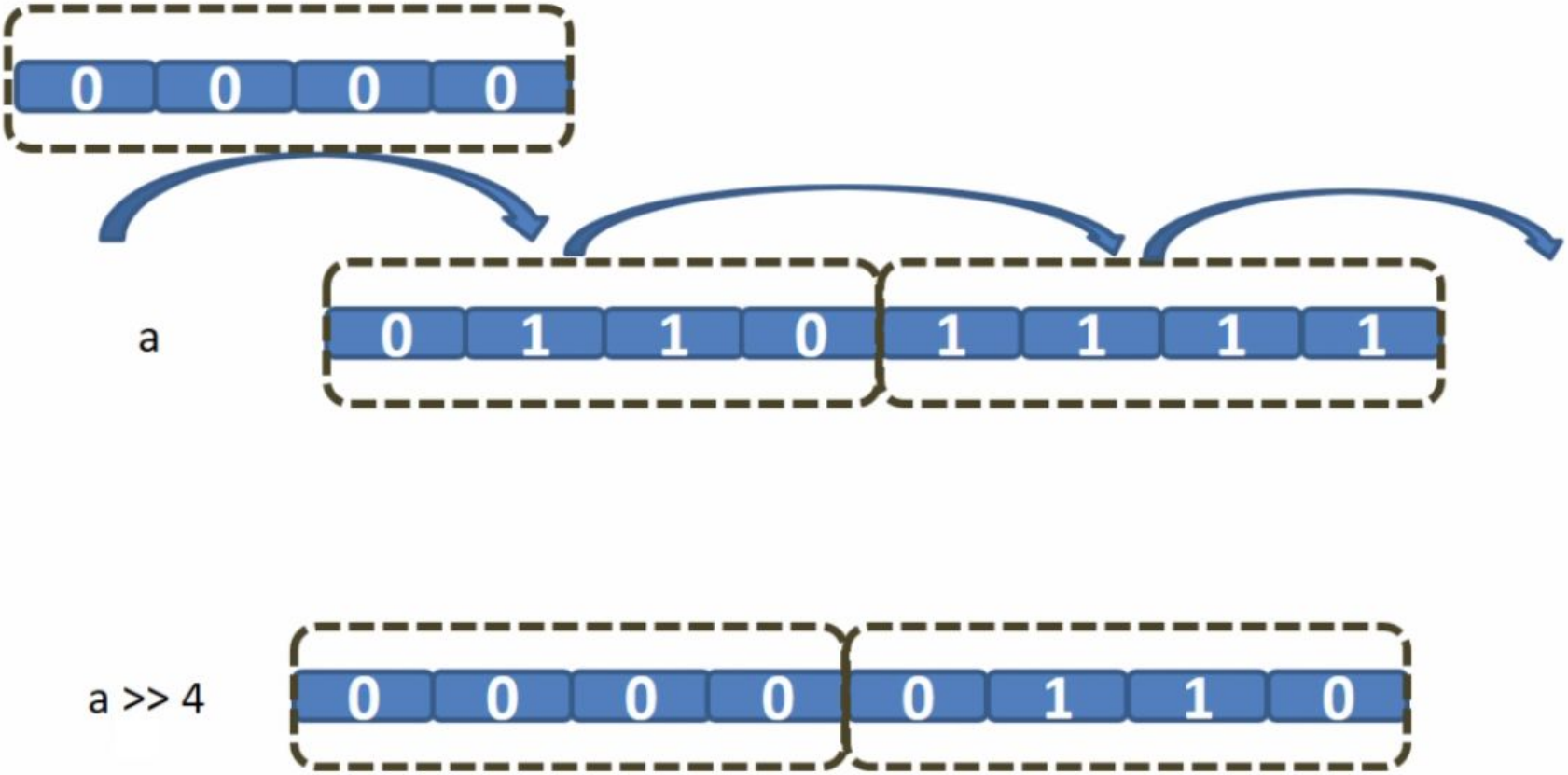
char a = 111

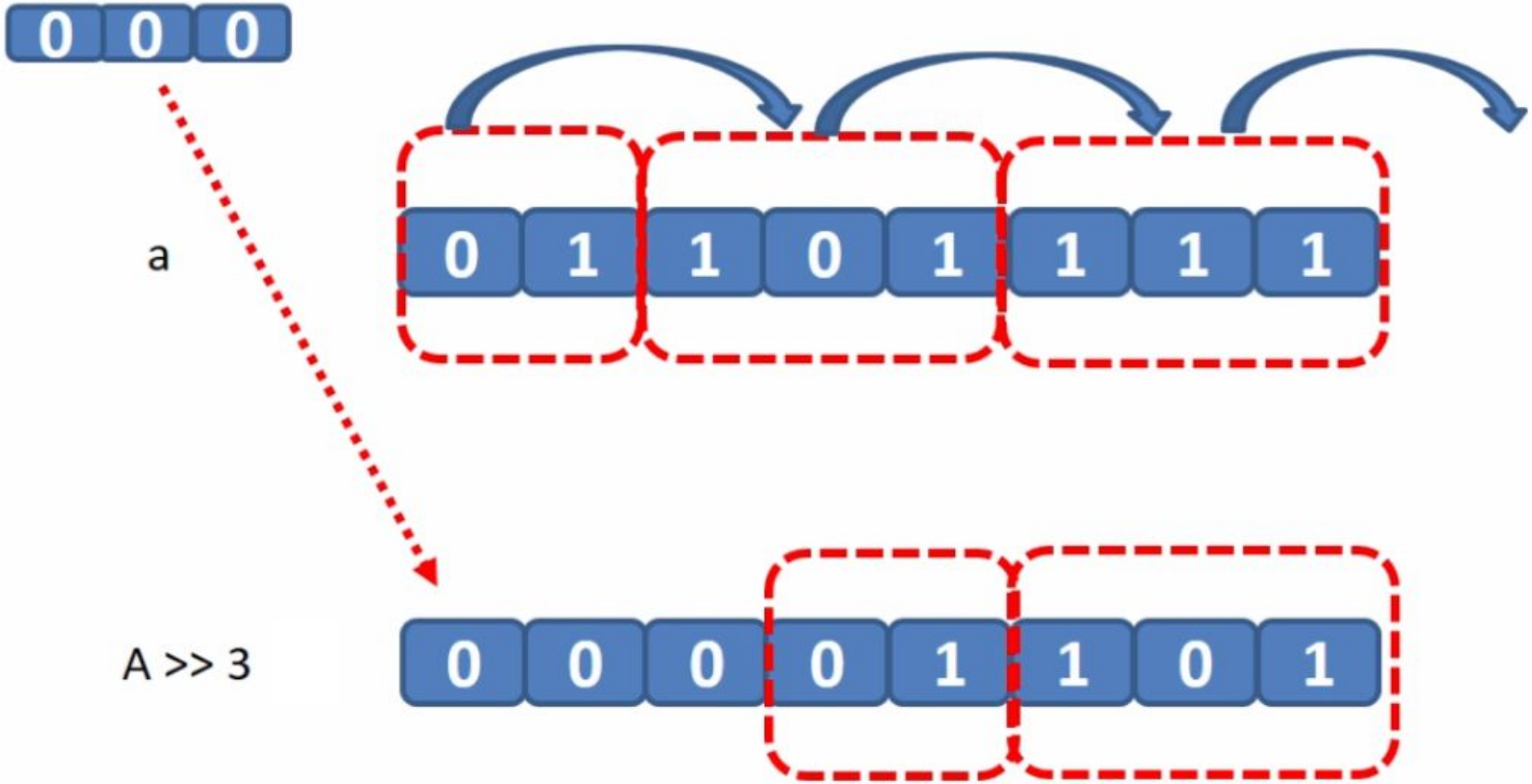
char b = a >> 4

b = ?


Now, ‘a’ must be shifted by 4 times to the right


0
a
0 1 1 0 1 1 1 1
111 (0x6F)
0
a >> 1
0 0 1 1 0 1 1 1
0
a >> 2
0 0 0 1 1 0 1 1
0
a >> 3
0 0 0 0 1 1 0 1
a >> 4
0 0 0 0 0 1 1 0
6 (0X06)


0 0 0 0
a
0 1 1 0 1 1 1 1
a >> 4
0 0 0 0 0 1 1 0

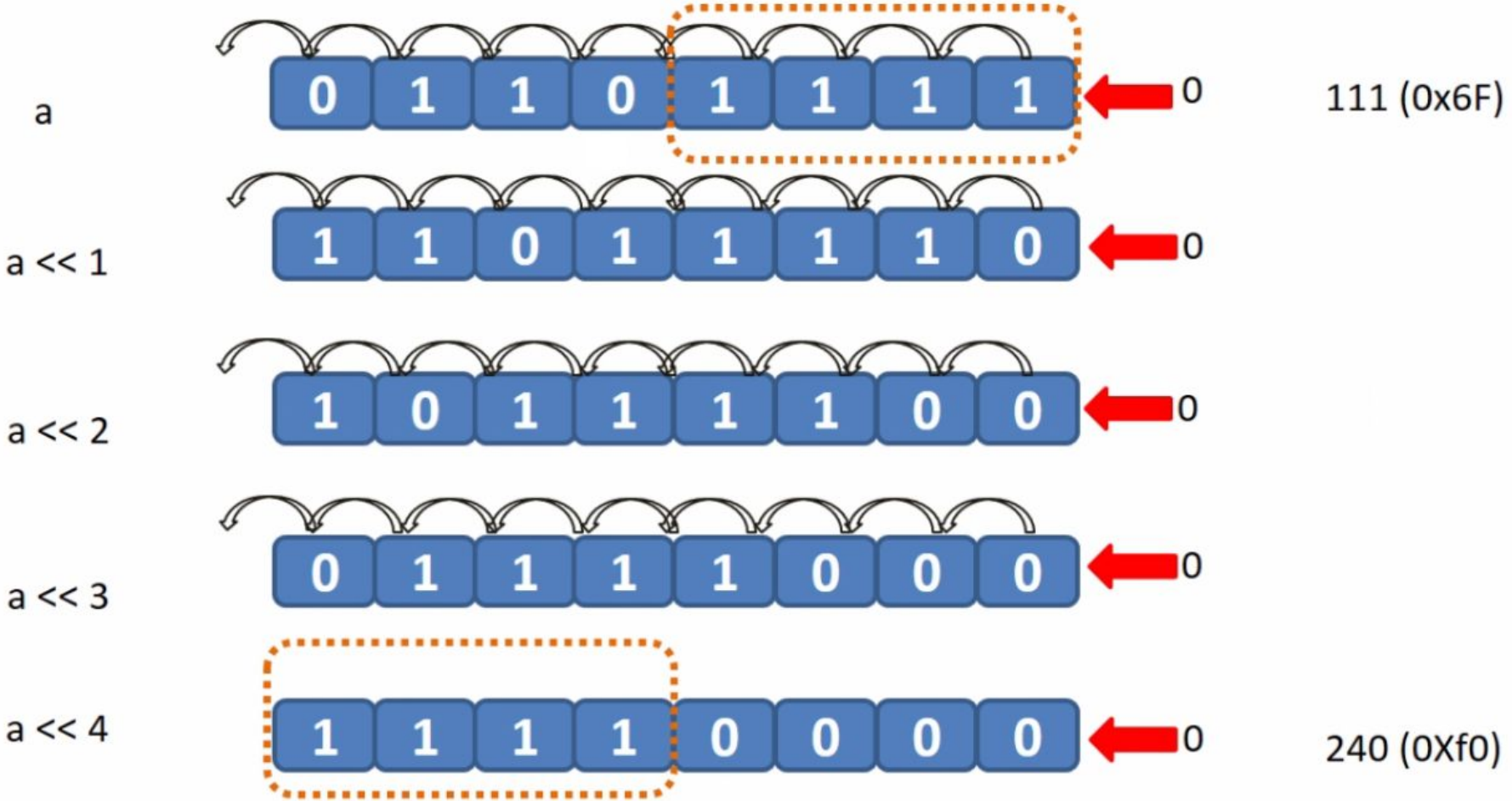

0 0 0
a
0 1 1 0 1 1 1 1
A >> 3
0 0 0 0 1 1 0 1

# Bitwise Operators in ‘C’

| | |
|---|---|
| & (bitwise AND) | >> (bitwise Right Shift) |
| \| (bitwise OR) | Unary ~ (bitwise NOT) (Negation) |
| << (bitwise Left Shift) | ^ (bitwise XOR) |

# Bitwise left shift operator(<<)

- This operator takes 2 operands
- Bits of the 1st operand will be left shifted by the amount decided by the 2nd operand
- Syntax : operand1 << operand2
- Lets see an example


a
0 1 1 0 1 1 1 1
0
111 (0x6F)
a << 1
1 1 0 1 1 1 1 0
0
a << 2
1 0 1 1 1 1 0 0
0
a << 3
0 1 1 1 1 0 0 0
0
a << 4
1 1 1 1 0 0 0 0
0
240 (0Xf0)

# << vs >>

| << (left shift) | >> (right shift) |
|---|---|
| 4 << 0 => 4 | 128 >> 0 => 128 |
| 4 << 1 => 8 | 128 >> 1 => 64 |
| 4 << 2 => 16 | 128 >> 2 => 32 |
| 4 << 3 => 32 | 128 >> 3 => 16 |
| 4 << 4 => 64 | 128 >> 4 => 8 |
| 4 << 5 => 128 | 128 >> 5 => 4 |
| | |

**A value will be multiplied by 2 for each left shift**
**A value will be divided by 2 for each right shift**

# Applicability of bitwise shift operations in embedded programming code

# Applicability

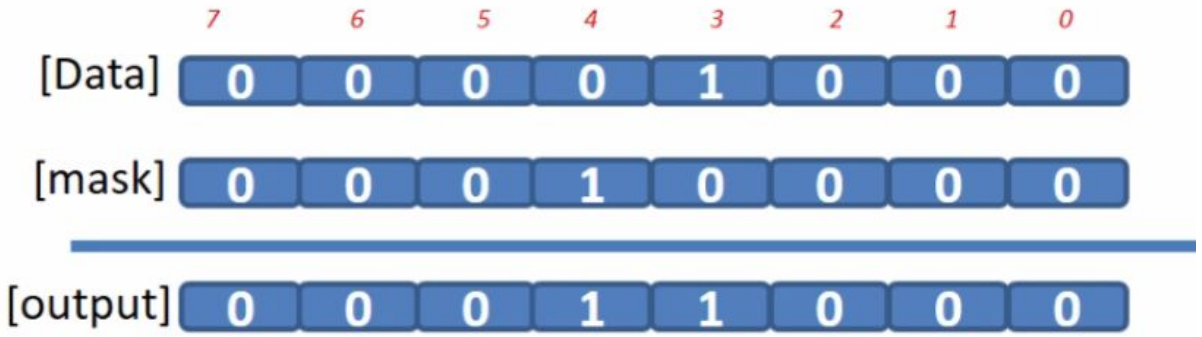
- Bitwise shift operators are very much helpful in bit masking of data along with other bitwise operators
- Predominantly used while setting or clearing of bits
- Lets consider this problem statement :Set $4^{th}$ bit of the given data

Set 4th bit of the given data

Data = 0x08

Data = Data | 0x10

= 0x18

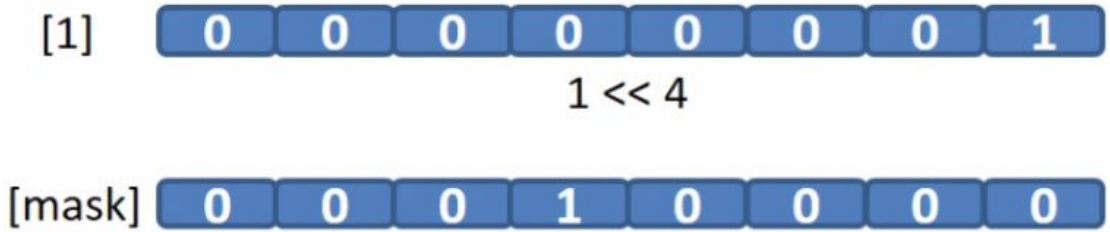



Set 4th bit of the given data

Data = 0x108

Data = Data | (1 << 4)

= 0x18

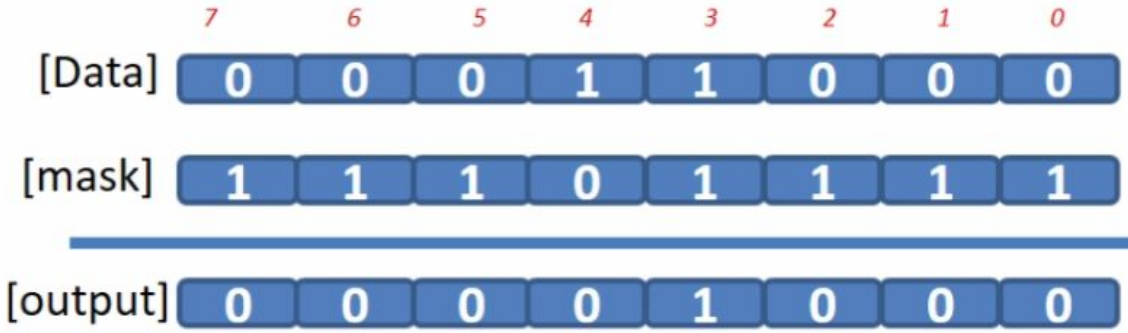


Clear 4th bit of the given data

Data = 0x18

Data = Data & 0xEF

= 0x08

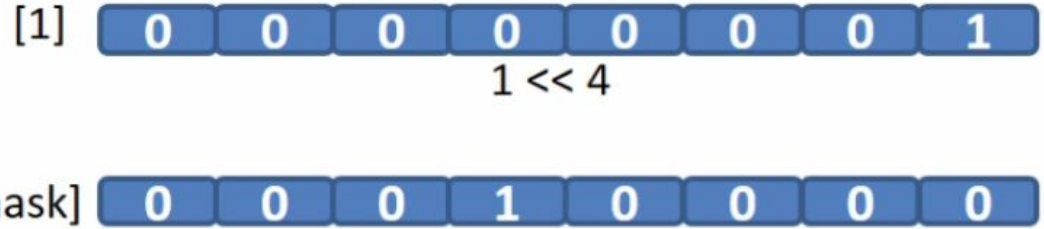



Clear 4th bit of the given data

Data = 0x18

Data = Data & ~(1 << 4)

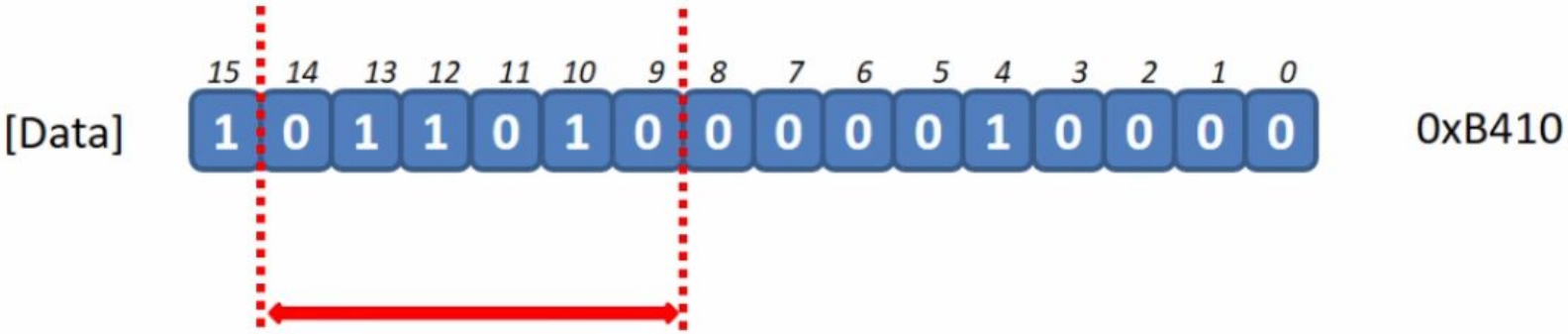
= 0x08



# Bit extraction

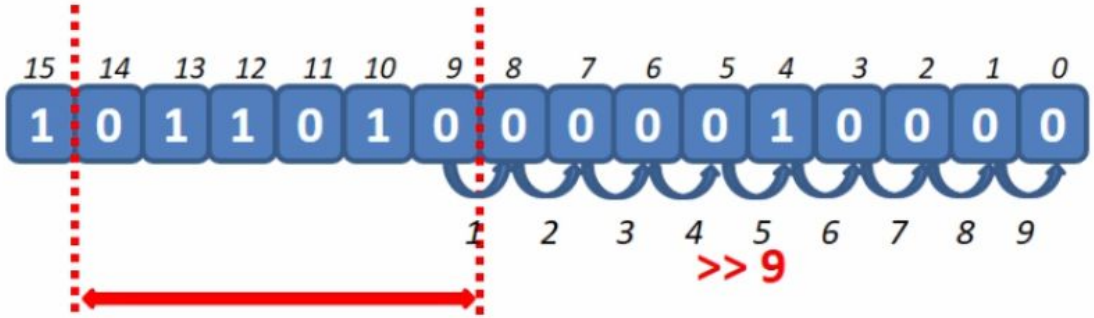
- Lets consider this problem statement

Extract bit positions from $9^{th}$ to $14^{th}$ [14:9] in a given data and save it in to another variable .

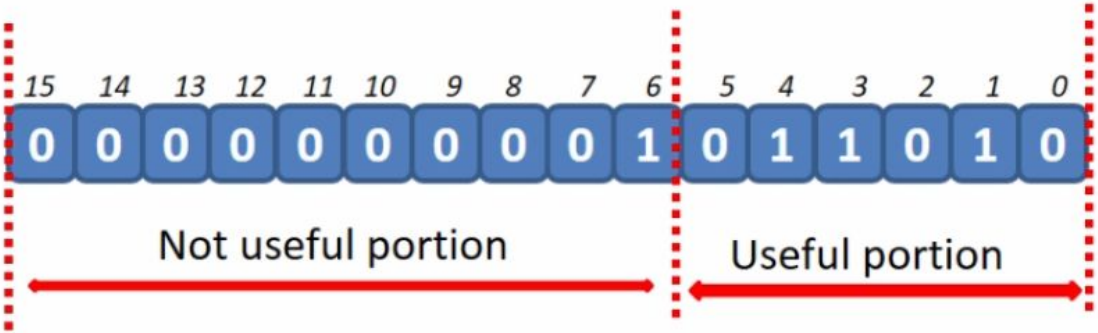



This portion needs to be extracted and saved in to another variable

1) Shift the identified portion to right hand side until it touches the least significant bit ( $0^{th}$ bit)
2) Mask the value to extract only 6 bits [5:0] and then save it in to another variable

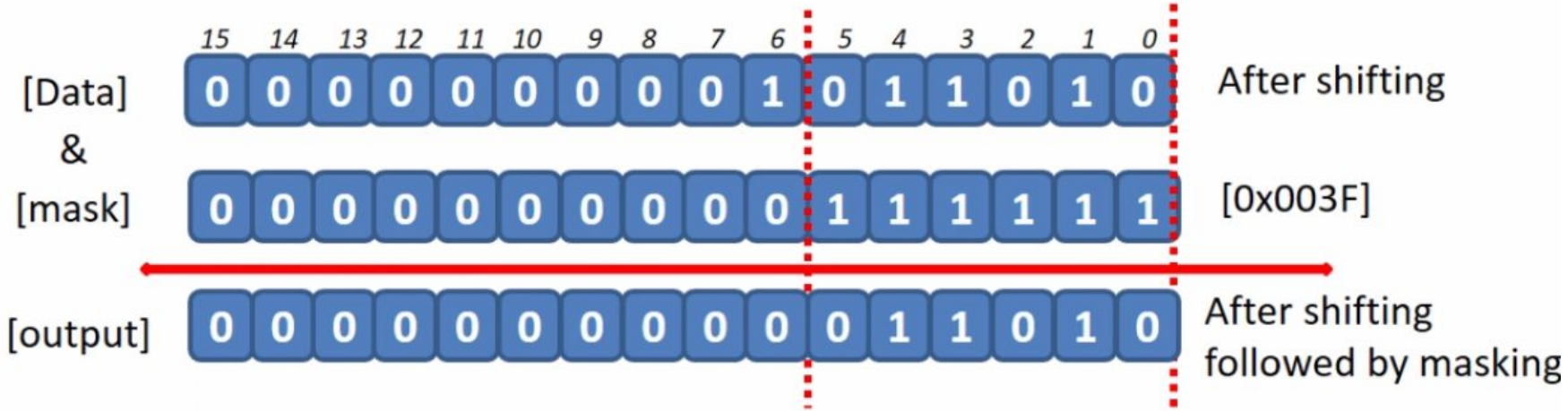



1) Shift the identified portion to right hand side until it touches the least significant bit ( $0^{th}$ bit)




Mask Data to retain useful portion and zero out not useful portion

2) Mask the new value to extract only first 6 bits[5:0] and then save it in to another variable



```
uint16_t Data = 0xB410;
uint8_t output ;
```

```
output = (uint8_t) ((Data >> 9) & 0x003F)
```

# Pre-processor Directives

# *Pre-processor Directives*

- In C programming pre-processor directives are used to affect compile-time settings
- Pre-processor directives are also used to create macros used as a textual replacement for numbers and other things.
- Pre-processor directives begin with the “#” symbol.
- Pre-processor directives are resolved or taken cared during the pre-processing stage of compilation

# *Pre-processor Directives supported in 'C'*

| Objective | Syntax of pre-processor directive |
|---|---|
| Macros | #define <identifier> <value><br>Used for textual replacement |
| File inclusion | #include <std lib file name><br>#include "user defined file name"<br>Used for file inclusion |
| Conditional compilation | #ifdef , #endif, #if , #else, #ifndef, #undef<br>Used to direct the compiler about code compilation |
| Others | #error #pragma |

# Macros in ‘C’ ( #define )

- Macros are written in ‘C’ using **#define** pre-processor directive
- Macros are used for textual replacement in the code, commonly used to define constants.
- Syntax : #define <Identifier> <value>
- Example : #define MAX_RECORD 10

# Use case

```c
#include <stdio.h>
#include <stdint.h>

int main(void)
{
    uint8_t age = get_voter_age();

    if(age < 18)
    {
        printf("you cannot vote !\n");
    }
    else
    {
        printf("Congratulations ! you can vote\n");
    }
}
```

```c
#include <stdio.h>
#include <stdint.h>

//This is a macro which defines minimum age for evaluation
#define MIN_AGE 18

int main(void)
{
    uint8_t age = get_voter_age();

    if(age < MIN_AGE)
    {
        printf("you cannot vote !\n");
    }
    else
    {
        printf("Congratulations ! you can vote\n");
    }
}
```

**During pre-processing stage of the compilation process, macro names(identifiers ) are replaced by the associated values inside the program.**

```c
#include <stdio.h>
#include <stdint.h>

//This is a macro which defines minimum age for evaluation
#define MIN_AGE 18

int main(void)
{
    uint8_t age = get_voter_age();

    if(age < MIN_AGE)
    {
        printf("you cannot vote !\n");
    }
    else
    {
        printf("Congratulations ! you can vote\n");
    }
}
```

**This is NOT a variable . This is an identifier (macro name)**
**Just a textual replacement for a number**

In Embedded system programming , we use lots of ‘C’ macros to define pin numbers, pin values, crystal speed, peripheral register addresses, memory addresses and for other configuration values.

```
#define PIN_8                   8
#define GREE_LED                PIN_8
#define LED_STATE_ON            1
#define Led_state_off           0
#define XTAL_SPEED              8000000UL
#define FLASH_BASE_ADDR         0x08000000UL
#define SRAM_BASE_ADDR          0x20000000ul

#define PI      3.1415;
```

## *function-like macros*

To define a function-like macro, use the same '#define' directive, but put a pair of parentheses immediately after the macro name.

```
#define PI_VALUE  3.1415

#define AREA_OF_CIRCLE(r)  PI_VALUE * r * r

areaCircle = AREA_OF_CIRCLE(radius);


areaCircle = PI_VALUE * radius * radius;

areaCircle = 3.1415 * radius * radius;
```

Original ‘C’ statement

Processed by pre-processor

This macro is poorly written and dangerous

```
#define AREA_OF_CIRCLE(r)   PI_VALUE * r * r
```

You have to be careful with macro ‘values’ when you are doing some “operations” using multiple “operands.”

```
#define PI_VALUE  3.1415

#define AREA_OF_CIRCLE(r)  PI_VALUE * r * r
```

```
areaCircle = AREA_OF_CIRCLE(radius+1);
```

Original ‘C’ statement

```
areaCircle = 3.1415 * radius+1 * radius+1
```

Processed by pre-processor

This macro is poorly written and dangerous

```
#define AREA_OF_CIRCLE(r)   PI_VALUE * r * r
```

```
#define AREA_OF_CIRCLE(r)       ( (PI_VALUE) * (r) * (r) )
```

# Best practises while writing Macros in ‘C’

1. Use meaningful macro names
2. It's recommended that you use UPPER case letters for macro names to distinguish them from variables.
3. Remember, macro's names are not variables. They are labels or identifiers, and they don't consume any code space or ram space during compile time or run time of the program.
4. make sure that parentheses surround the macro value
5. While using function-like macros or when you are using macros along with any operators, always surround the operands with parentheses.

# Conditional compilation using pre-processor directives

# Conditional compilation in ‘C’

#if
#ifdef
#endif
#else
#undef
#ifndef

Example of condition compilation directives.

These directives help you to include or exclude individual code blocks based on various conditions set in the program

# #if and #endif directive

Syntax :

#if <constant expression>

#endif

Example :

#if 0

//code block

#endif

This directive checks whether the constant expression is zero or non zero value.
If constant is 0, then the code block will note be included for the code compilation.
If constant is non zero, the code block will be included for the code compilation.

***#endif*** *directive marks the end of scope of #if , #ifdef, #ifndef , #else, #elif directives*

# #if ...#else .....#endif

Syntax :

```
#if  <constant expression>

#else

#endif
```

Example :

```
#if   1
//Code block-1
#else
//Code block-2
#endif
```

# #ifdef

Synatx :

```
#ifdef  <identifier>


#endif
```

Example:

```
#ifdef  NEW_FEATURE
//Code block
#endif
```

#ifdef directive checks whether the identifier is defined in the program or not. If the identifier is defined, the code block will be included for compilation.

2. Execute circle are calculation code block only if AREA_CIR macro is defined and execute triangle area calculation code block only if AREA_TRI macro is defined in the program

3. Execute circle area code block only if AREA_TRI is not defined

# #ifndef

Synatx :

```
#ifndef  <identifier>


#endif
```

Example :

```
#ifndef  NEW_FEATURE
//Code block for old feature
#endif
```

#ifndef directive checks whether the identifier is defined in the program or not. If the identifier is not defined then the code block will be included for compilation.

# #ifdef and #else

```
#ifdef NEW_FEATURE
//Code block of new feature
#else
//code block for old feature
#endif
```

# #if and ‘defined’ operator

- The defined operator is used when you want to check definitions of multiple macros using single #if, #ifdef or #ifndef directives
- You can also use ‘C’ logical operators such as AND, NOT, OR with 'defined' operator

Thank You