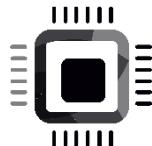


Microcontroller Embedded ‘C’ Programming : Absolute Beginners

FASTBIT EMBEDDED BRAIN ACADEMY

www.fastbitlab.com

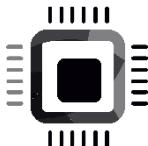


License

This power point presentation by BHARATI SOFTWARE is licensed under [CC BY-SA 4.0](#)

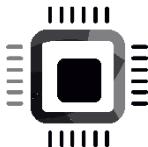
To view a copy of this license, visit

<https://creativecommons.org/licenses/by-sa/4.0>



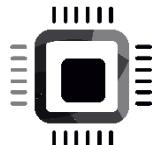
Important Links

- For the full video course please visit
 - <https://www.udemy.com/course/embedded-system-design-using-uml-state-machines/>
- Course repository
 - <https://github.com/niekiran/EmbeddedUMLStateMachines>
- Explore all FastBit EBA courses
 - <http://fastbitlab.com/course1/>
- For suggestions and feedback
 - contact@fastbitlab.com



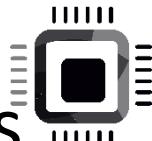
Social media

- Join our Facebook private group for technical discussion
 - <https://www.facebook.com/groups/fastbiteba/>
- LinkedIn
 - <https://www.linkedin.com/company/fastbiteba/>
- Facebook
 - <https://www.facebook.com/fastbiteba/>
- YouTube
 - <https://www.youtube.com/channel/UCa1REBV9hyrzGp2mjJCagBg>
- Twitter
 - <https://twitter.com/fastbiteba>



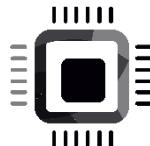
What is a program ?

- A program is a series of instructions that cause a computer or a microcontroller to perform a particular task.
- A computer program includes more than just instructions. It also contains data and various memory addresses on which the instructions work to perform a specific task.



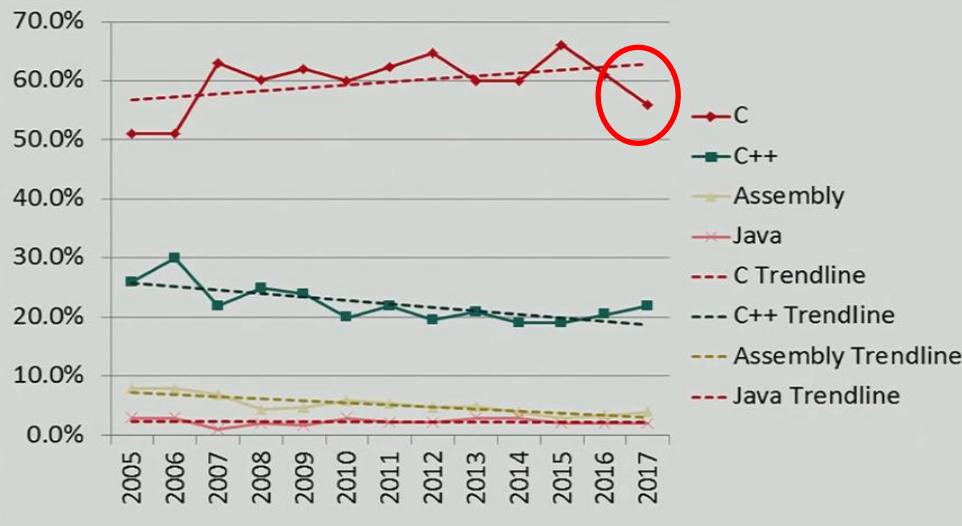
Programming languages in Embedded Systems

- C and C++ (most widely used)
- Rust
- Assembly
- Java
- Python



Survey by Embedded.com

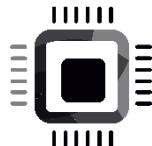
It's Mostly C, Some C++, and Not Much Else



Dan Saks
Writing better
embedded Software

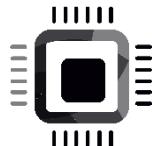


Meeting Embedded 2018



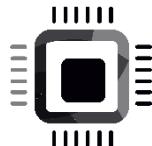
C programming standardization

- The C programming language was initially developed in AT&T labs by Professor Brian W. Kernighan and Dennis Ritchie, known as K&R C.
- During the year 1970, ‘C’ programming became very popular but without any serious standardization to the language.
- The non-official standard was called K&R C, which led to many ambiguities among different compiler programmers, and that led to non-portable codes.
- K&R C was the first non-official ‘C’ standard



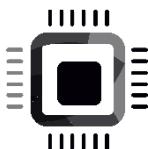
C programming standardization

- In 1989 American National Standard Institute designed and approved first official C standard called X3.159-1989, and in 1990 it was approved by ISO as an international standard for C programming language: ISO/IEC 9899:1990 (This is also called ANSI C or C89 or C90 standard in short)
- The language underwent few more changes (addition of new features, syntaxes, data types, etc.) and newly updated standard released in 1999 under the ISO tag ISO/IEC 9899:1999 which is also called C99 standard in short



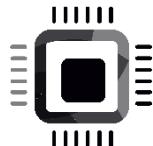
C programming standardization

- C11 is an informal name for ISO/IEC 9899:2011, which is a new standard approved in December 2011. C11 supersedes the C99 standard.
- In this course, we will be using the ‘C11’ standard with some compiler (GCC) extensions (gnu11). More on this, we will see later.



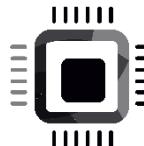
Compatibility

- Note that if you have written a code using C90 standard, then it will compile without any issues in `-std=C99` compilation(backward compatibility)
- If you have written a code using C99 standard specific features, then It may not compile successfully in C90 compilation



First ‘C’ Program “Hello World”

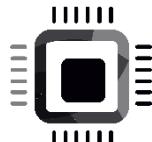
Let’s write a ‘C’ code which simply displays the text **Hello world** on the “console” and exits.



Escape Sequences

Meaning

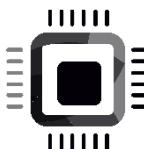
\'	Single Quote
\"	Double Quote
\\\	Backslash
\0	Null
\a	Bell
\b	Backspace
\f	form Feed
\n	Newline
\r	Carriage Return
\t	Horizontal Tab
\v	Vertical Tab



Difference between \n and \r

\n : moves the cursor to the beginning of the new line vertically

\r : moves the cursor to the beginning of the current line horizontally



CURSOR

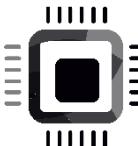
Line 1

main.c

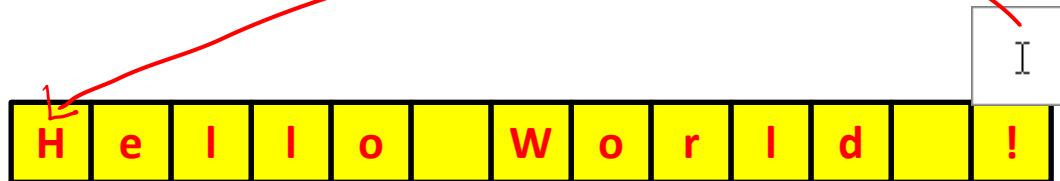
```
1 #include <stdio.h>
2
3 int main()
4 {
5     printf("Hello World !\n");
6     printf("bye !");
7 }
8
9
10
```

bye ! World !

...Program finished with exit code 0
Press ENTER to exit console.



'r' moves the cursor horizontally to the beginning of the current line



Line 1

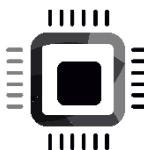
The screenshot shows a terminal window with the following content:

```
main.c
1
2 #include <stdio.h>
3
4 int main()
5 {
6     printf("Hello World !\n");
7     printf("bye !");
8 }
9
10
```

At the bottom of the terminal window, the output is displayed:

```
bye ! World !

...Program finished with exit code 0
Press ENTER to exit console.
```



cursor

b y e ! W o r l d !

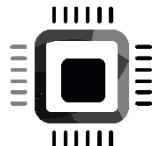
Line 1

main.c

```
1 #include <stdio.h>
2
3 int main()
4 {
5     printf("Hello World !\n");
6     printf("bye !");
7
8 }
9
10
```

bye ! World !

...Program finished with exit code 0
Press ENTER to exit console.



Check your understanding

Write a program to print the below texts using printf function

You can download and copy the texts from resource section of this lecture.

David says , " Programming is fun !"

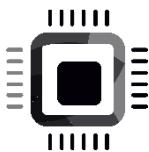
**Conditions apply , "Offers valid until tomorrow"

C:\My computer\My folder

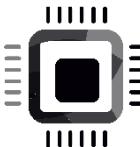
D:/My documents/My file

\ \ \ \ Today is holiday \ \ \ \

This is a triple quoted string """ This month has 30 days """



Overview of a ‘C’ program



```
/* A simple 'C' program written by me */

#include <stdio.h>

//This is a global variable declaration and initialization
int a = 10;
int b = 20;

int main(void)
{
    //This is a local(automatic) variable declaration
    int x;

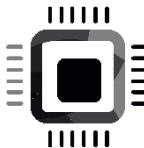
    printf("Input an integer\n");

    scanf("%d", &x); // %d is used for an integer

    //Printing the value of x to the console
    printf("The integer is: %d\n", x);

    //Printing the value of global variables a and b
    printf("value of the global variables: %d and %d\n", a,b);

    return 0;
}
```



```
/* A simple 'C' program written by me */

#include <stdio.h>

//This is a global variable declaration and initialization
int a = 10;
int b = 20;

int main(void)
{
    //This is a local(automatic) variable declaration
    int x;

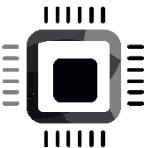
    printf("Input an integer\n");

    scanf("%d", &x); // %d is used for an integer

    //Printing the value of x to the console
    printf("The integer is: %d\n", x);

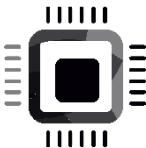
    //Printing the value of global variables a and b
    printf("value of the global variables: %d and %d\n", a,b);

    return 0;
}
```



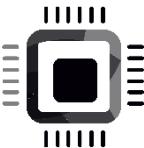
Comments

- ✓ Generally **Comments** are used to provide the description or documentation about the code you have written.
- ✓ Through commenting you can mention important note , briefly explain behaviour of the code, and other useful details.
- ✓ Comments help the developer understand the logic/algorithm of the code if he/she revisits it after a long time.



Comments

- ✓ You can mention the comments anywhere you want in the program and your comments will be ignored by the compiler .
- ✓ Comments do not affect program or they do not consume any program memory space . They are not part of the final executable generated . It is there just for the documentation purpose .



Comments

In C there are two types of comments :

Single line comment

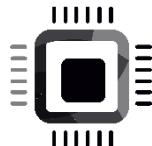
Multi-line comment

Single line comments

Single line comments can be provided by using `//.....`

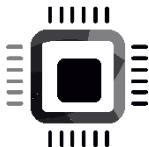
Multiple line comments

Multiple line comments can be provided by using `/*.....*/`



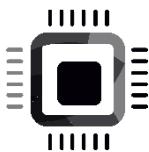
Multi line comments

```
/* C standard library header file inclusion
   This is comment line 1
   This is comment line 2
   This is comment line 3
   This is comment line 4      */
#include <stdint.h>
```

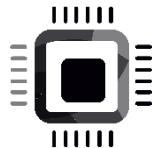


single line comments

```
// C standard library header file inclusion
// This is comment line 1
// This is comment line 2
// This is comment line 3
// This is comment line 4
#include <stdint.h>
```

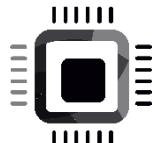


‘C’ data types and variables



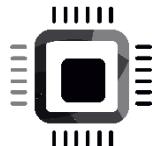
Data types :

- ✓ 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



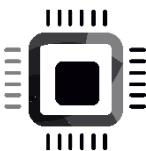
Representing real world data

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



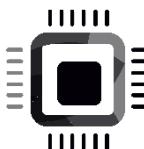
Examples

- 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



Data types :

Based on the real-world data we have
two significant data types in C



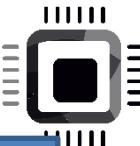
'C' data types

Integer data types

Used to represent whole numbers
(Integers)
Ex: 10, 20, 20 , etc

Float data types

Used to represent real
numbers
Ex : 10, 24.5,
60.000001 etc



Integer data types(for signed data)

char

Short int¹

int

long int²

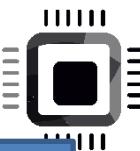
long long int³

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

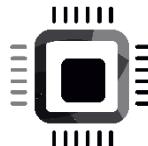
unsigned short int

unsigned int

unsigned long int

unsigned long long int

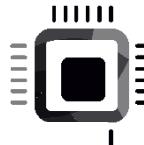
'C' integer data types , their storage sizes and value ranges



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

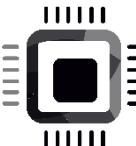
Meaning of memory size :

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

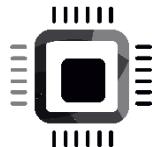
Data Type	Size in Bits	Data Range/Usage
unsigned char	8-bit	0 to 255
char	8-bit	-128 to +127
unsigned int	16-bit	0 to 65,535
int	16-bit	-32,768 to +32,767
unsigned short	16-bit	0 to 65,535
short	16-bit	-32,768 to +32,767
unsigned short long	24-bit	0 to 16,777,215
short long	24-bit	-8,388,608 to +8,388,607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



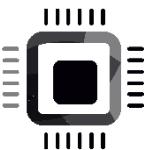
armcc cross compiler for 32 bit ARM based microcontrollers

Type	Size in bits	Natural alignment	Range of values
char	8	1 (byte-aligned)	0 to 255 (unsigned) by default. -128 to 127 (signed) when compiled with <code>--signed_chars</code> .
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

'C' integer data types , their storage sizes and value ranges



- ✓ The C standard does not fix the storage sizes of different data types. It only talks about the minimum and maximum values .
- ✓ For example C standard says, the minimum storage size of a long type variable is 32 bits, and the max is 64 bits. So, exact size of the long type variable depends on the compiler design.
- ✓ Some compilers fix 32 bits storage size for long type variables and some compilers fix 64 bits. Same is true for int data type, Some compilers fix 16bits storage size for int type variables and some compilers fix 32 bits.

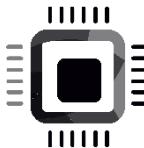


These data types will always be fixed size irrespective of compilers

Short(singed or unsigned) is always 2 bytes

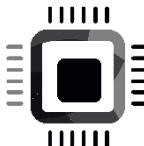
char(singed or unsigned) is always 1 byte

long long(singed or unsigned) is always 8 bytes



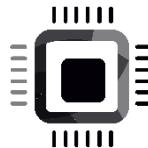
Integer data type : char

- ✓ 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

- **char** range : -128 to 127
 - A **char** data type will be used to store 1 byte of signed data
- **Unsigned char** range : 0 to 255
 - An **unsigned char** data type will be used to store 1 byte of unsigned data



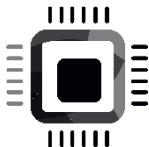
I want to store the current temperature data of city 'X' in my program.

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.

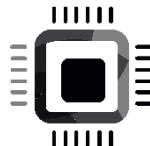
So, here we can use **unsigned char** data type to represent temperature value.



Example-1

```
unsigned char cityXTemperature;  
cityXTemperature = 25;
```

Here, **cityXTemperature** is a variable of type **unsigned char** to hold the integer value 25.



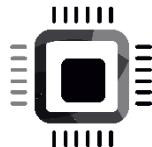
Variable definition

Data type Variable name

```
unsigned char cityXTemperature; /* This is a variable definition */  
cityXTemperature = 25; /* This is variable initialization */
```

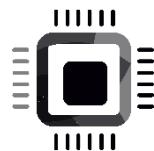
- ✓ **Variable definition always includes “data type ” followed by “variable name”**
- ✓ **Before using a variable , you should always define it using appropriate data type**
- ✓ **You can select the data type according your program logic and need .**

Example-2



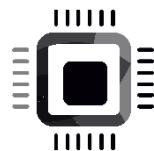
- ✓ Storing sun's temperature value in a program.
- ✓ Sun's surface temperature is 5,505 °C. Moreover, we know that it will never be –ve value for another few billion years :D.
- ✓ So we can consider it as unsigned data (+ve data) .
- ✓ $5505 > 255$. so, we cannot use **unsigned char**.
- ✓ So, in this case, we can safely use the **unsigned short int** data type like below.

```
unsigned short sunTemperature = 5005;
```

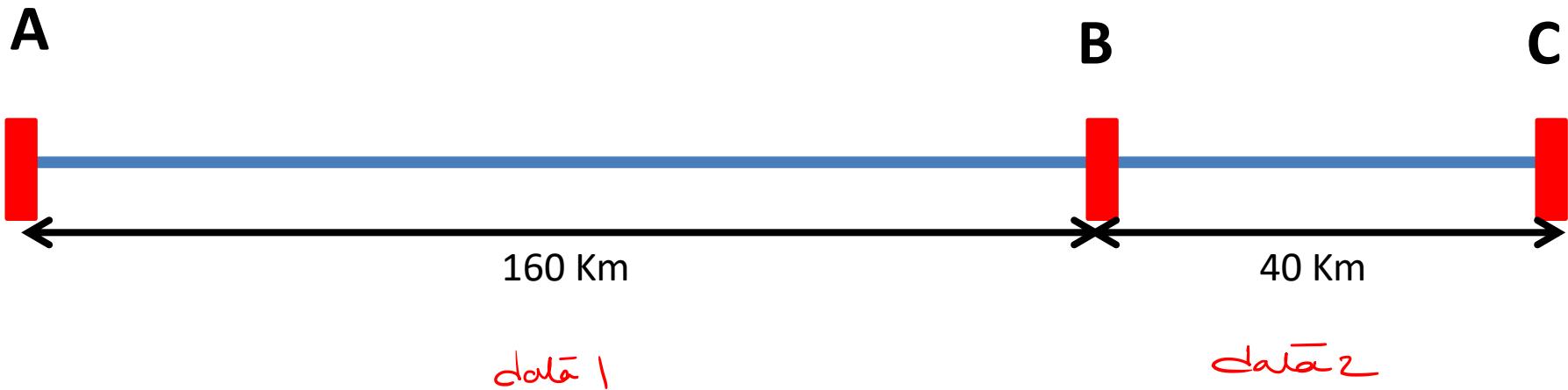


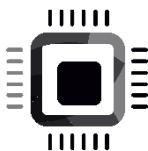
Write a C program to calculate and print distance from A to C





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

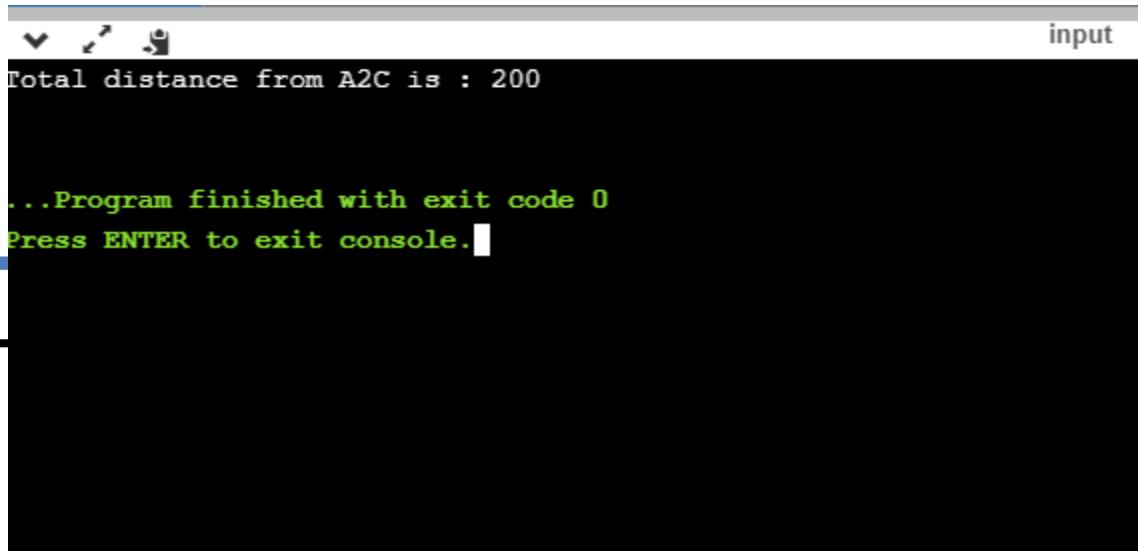




A

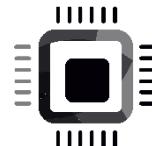


C



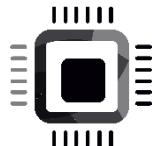
A screenshot of a terminal window titled "input". The window displays the following text:
Total distance from A2C is : 200

...Program finished with exit code 0
Press ENTER to exit console.



Format specifier

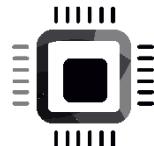
Format specifier	Description
%d	Integer Format Specifier
%f	Float Format Specifier
%c	Character Format Specifier
%s	String Format Specifier
%u	Unsigned Integer Format Specifier
%ld	Long Int Format Specifier



Range Calculation

Let us calculate the range of **char** data type.

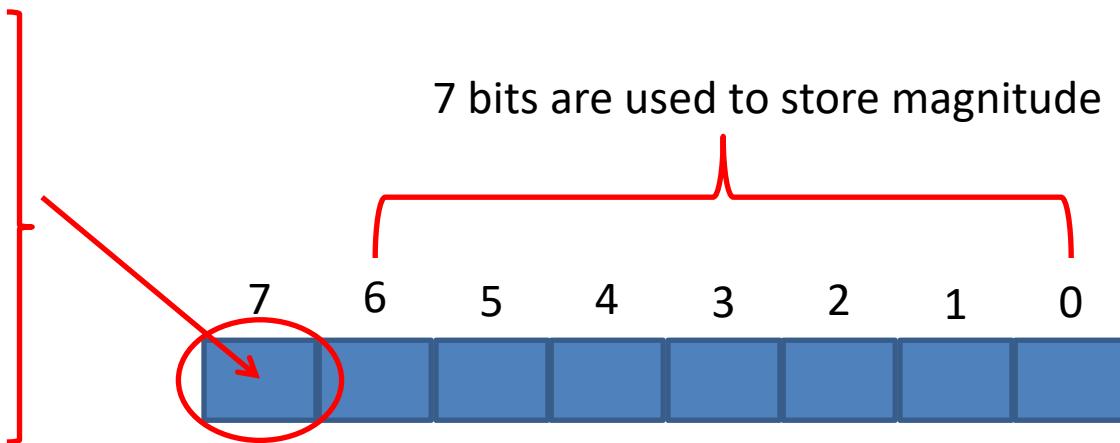
- ✓ We know that variable of the **char** data type consumes 1 byte of memory.
- ✓ Also, the **char** data type will be used to store 1 byte of signed data.



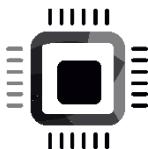
1 byte signed data representation

The most significant bit is used to represent the sign of the data

0 → data is +ve
1 → data is -ve

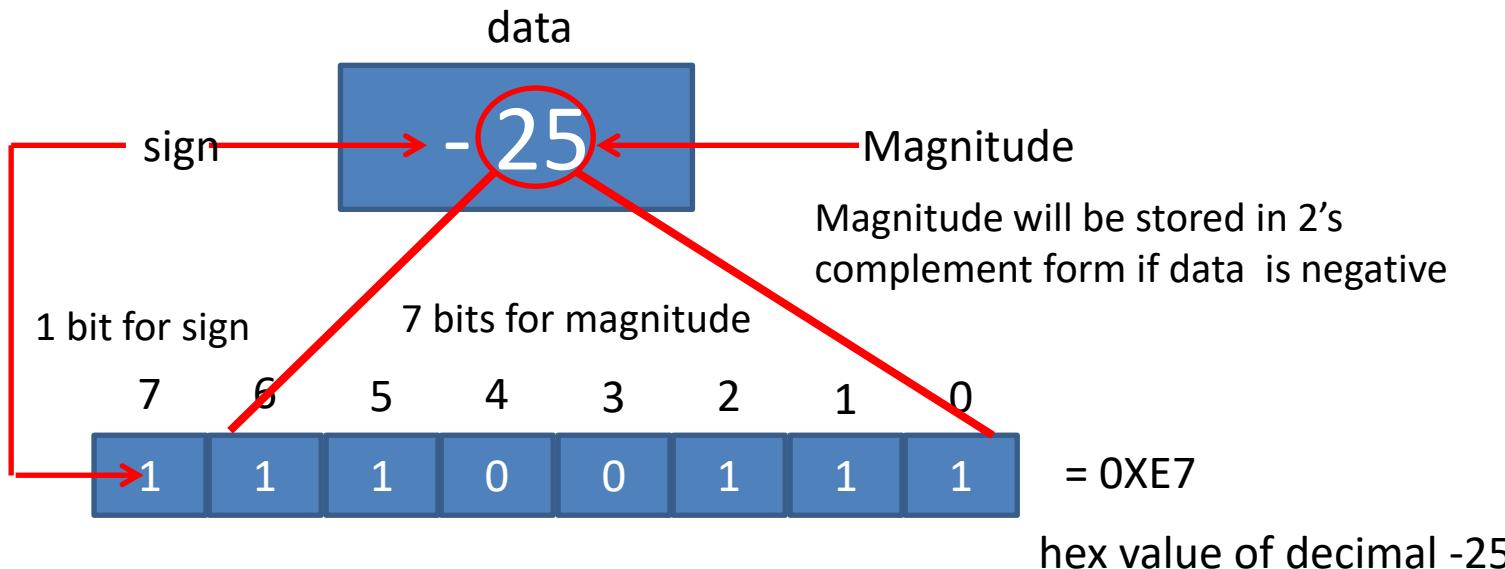


**Magnitude is stored in 2's complement format if data is negative.
Signed data means a dedicated 1 bit is used to encode a sign of the data.**

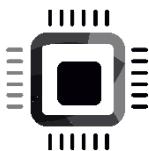


Example

Represent the data -25 in 1 byte signed data representation

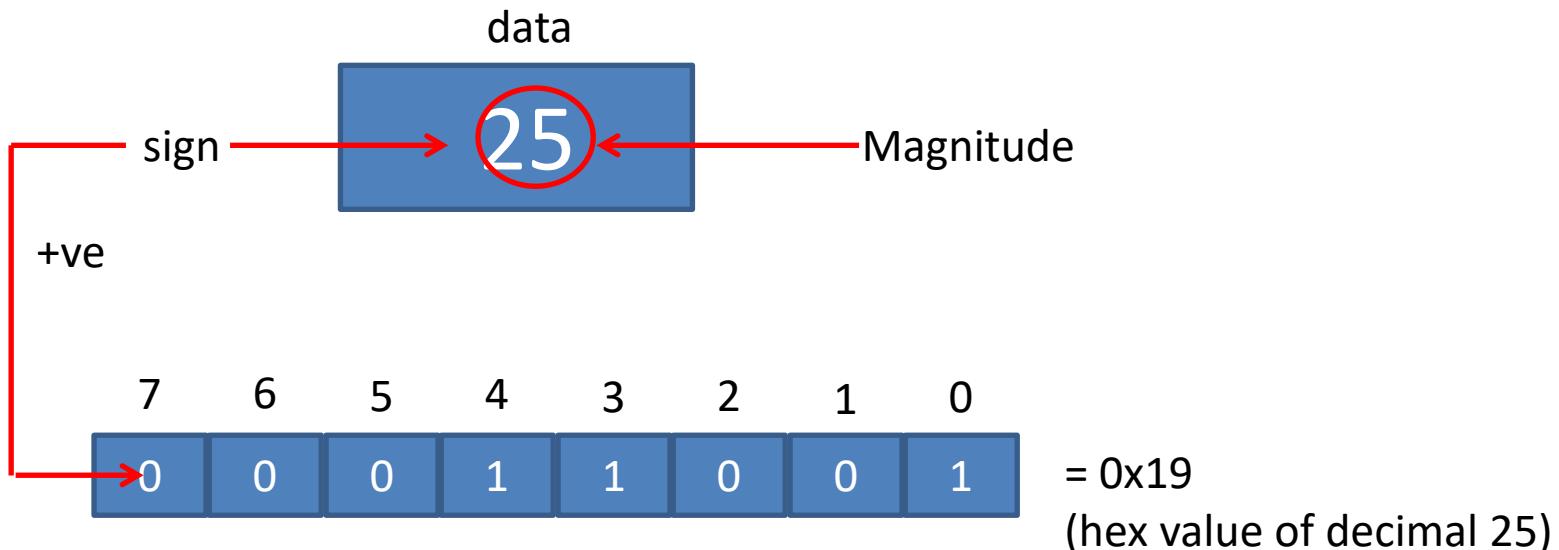


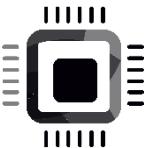
So, in signed data context 0xE7 is -25



Example

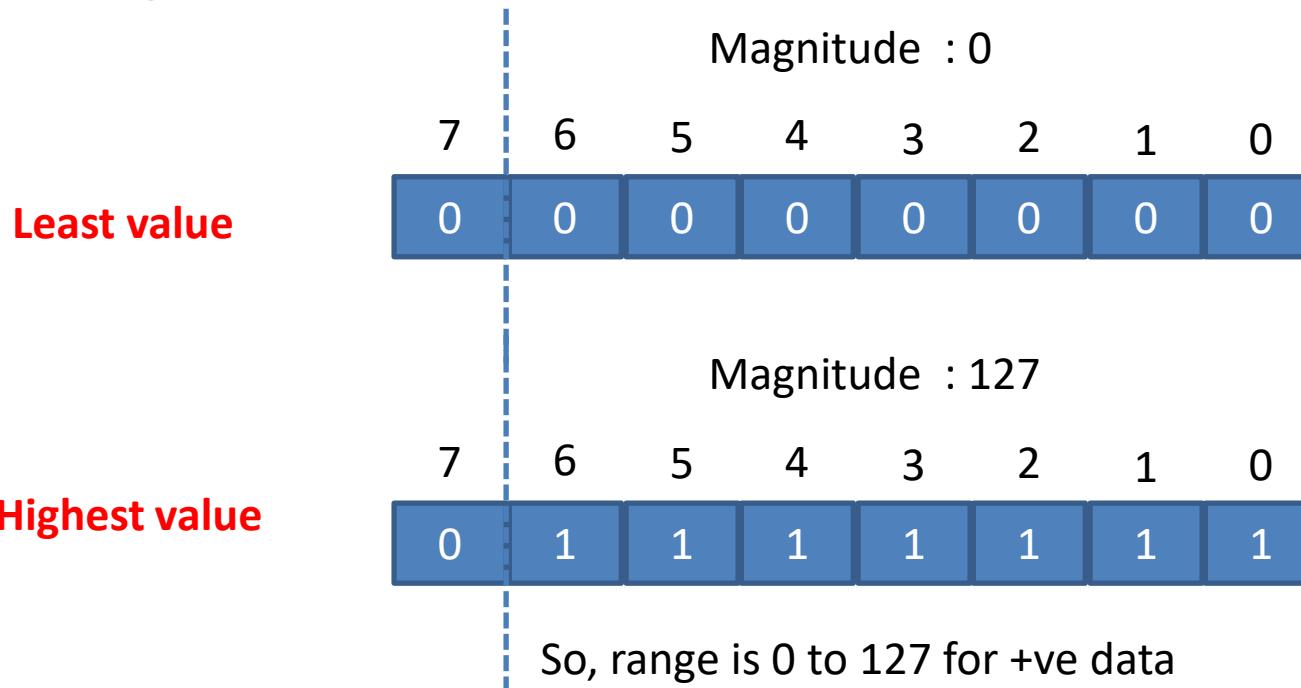
Represent the data 25 in 1 byte signed data representation

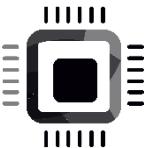




char data type range

When sign is +ve



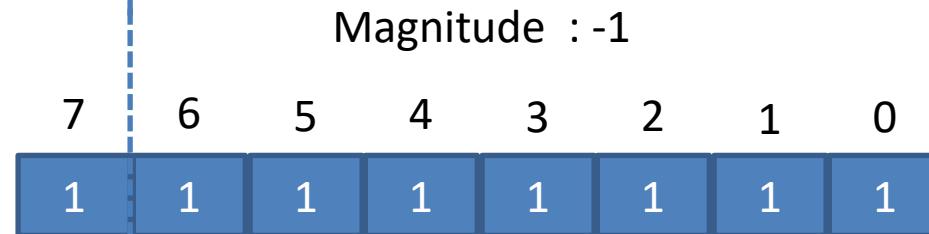


char data type range

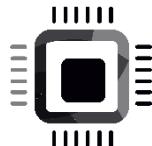
When sign is -ve



Highest value

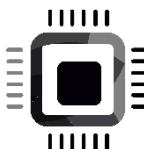


So, range is -128 to -1 for -ve values



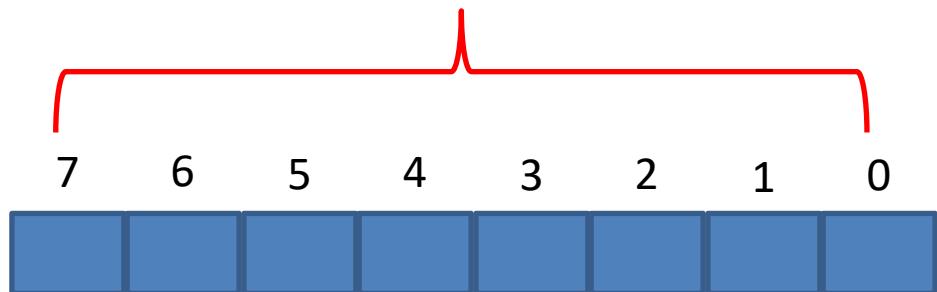
unsigned char range

- The **unsigned char** data type is used to store 1 byte of unsigned data.
- In unsigned data representation no particular bit is dedicatedly used to encode the sign.

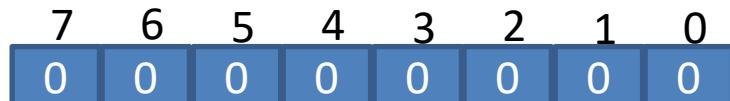


1 byte unsigned data representation

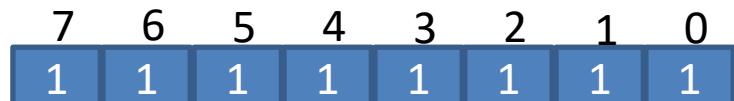
8 bits are used to store magnitude



Least value
0



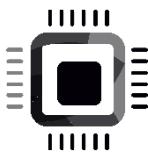
Highest value
255



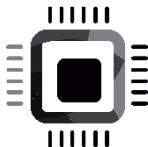


Integer data type : **short int** and **unsigned short int**

- ✓ Variable of type **short int** is used to store 2 bytes of signed data.
- ✓ Variable of type **unsigned short int** is used to store 2 bytes of unsigned data.
- ✓ You can just mention **short** (for signed) or **unsigned short** . “int” will be assumed.
- ✓ **short** type variable always consumes 2 bytes of memory irrespective of compilers.



Range calculation of **short int**



Represent the data -25 in 2 byte signed data representation

data

sign

25

Magnitude will be stored in 2's complement form
If data is negative

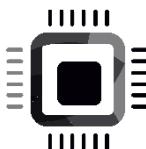
Magnitude

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	0	0	0	0	1	1	0	0	1

Binary representation of 25 using 2 bytes

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	1	1	1	1	1	0	0	1	1	0

1s complement format of 25



15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	1	1	1	1	1	0	0	1	1	0

1's complement format of 25

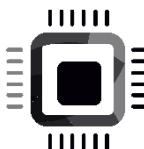
+

1

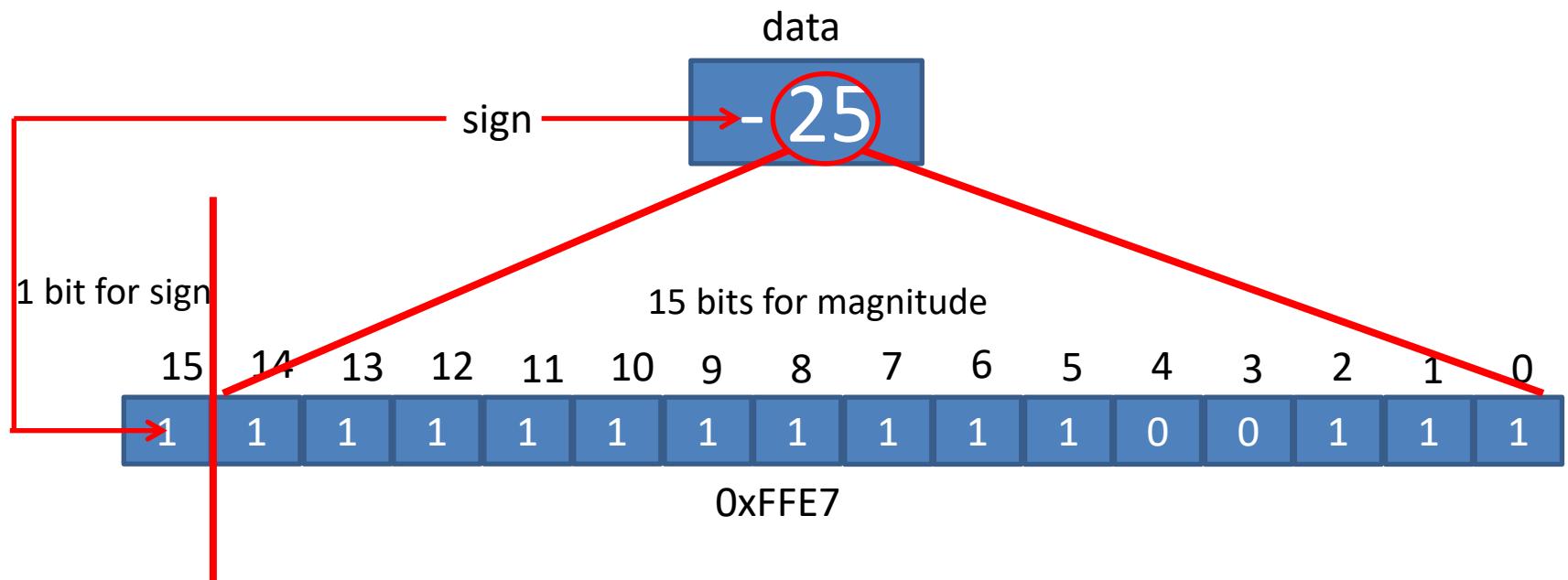
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	1	1	1	1	1	0	0	1	1	1

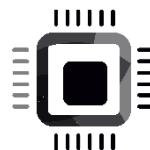
Sign bit

2's complement format representation of -25 using 2 bytes

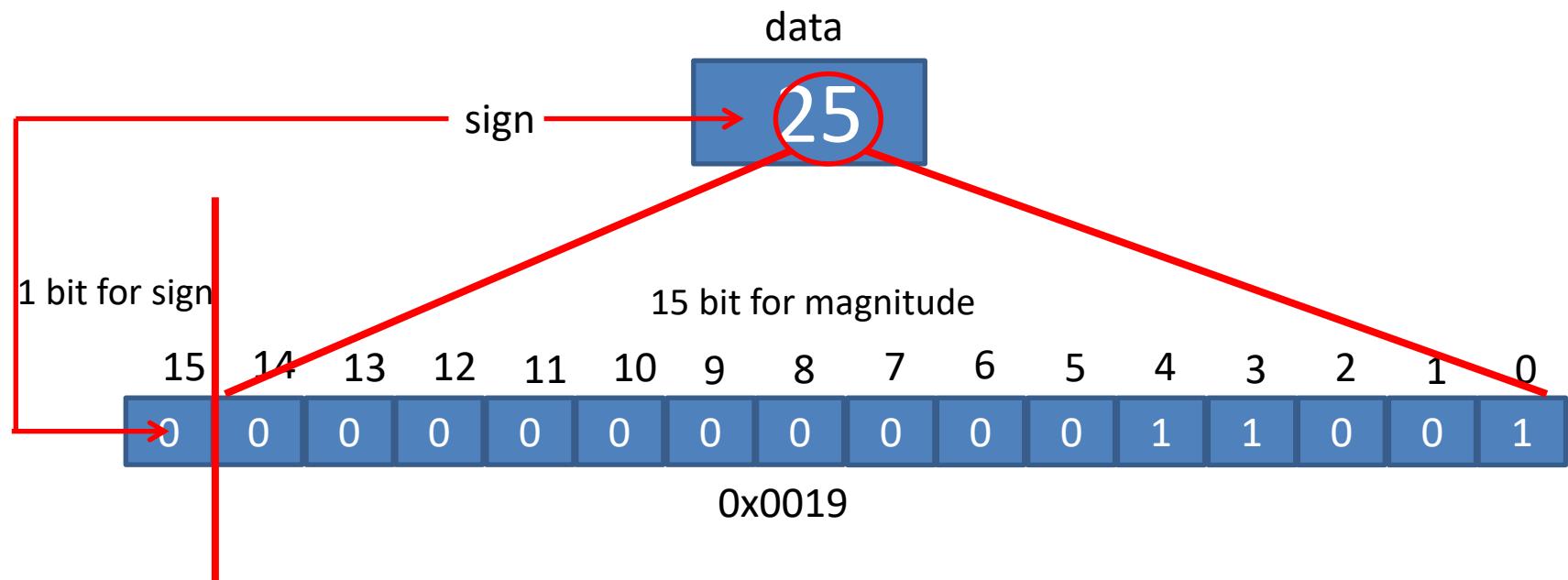


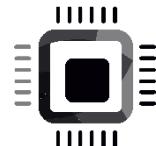
Represent the data -25 in 2 byte signed data representation





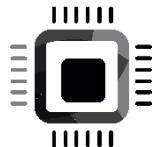
Represent the data 25 in 2 byte signed data representation



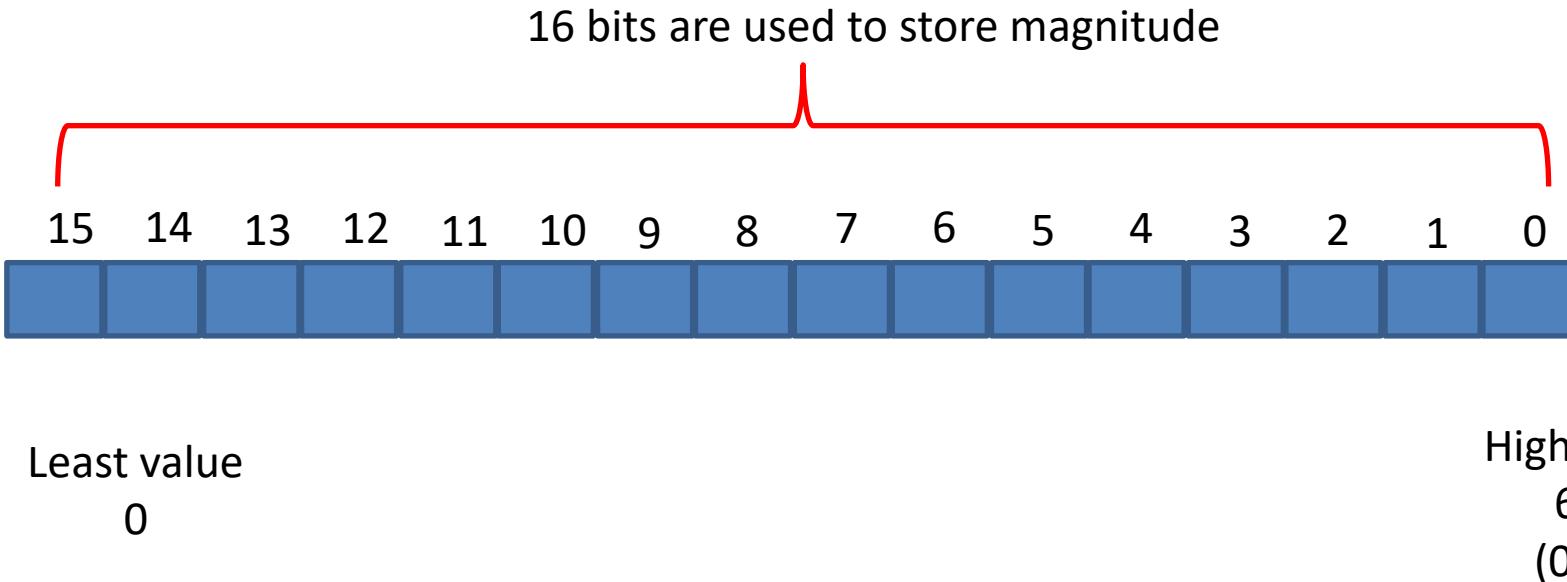


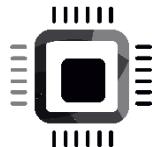
Range calculation of short int

- **short** range : -32,768 to 32,767
- **unsigned short** range : 0 to 65535



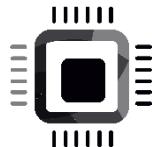
2 byte unsigned data representation





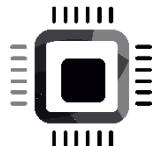
Integer data type : **int** and **unsigned int**

- ✓ **int** is an integer data type to store signed integer data
- ✓ An **int** type variable consumes 2 bytes of memory or 4 bytes of memory.
- ✓ Size of an **int** is decided by the compiler being used to generate code for your target hardware. i.e. you need to consult the compiler user manual to understand the size of an int. It is typically 2 or 4 bytes.
- ✓ **unsigned int** is an integer data type to store unsigned integer data.



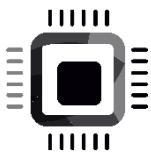
Integer data type : **long** and **unsigned long**

- ✓ **long** is an integer data type to store signed integer data
- ✓ A **long** type variable consumes 4 bytes of memory or 8 bytes of memory.
- ✓ Size of **long** data type is decided by the compiler being used to generate code for your target hardware. i.e. you need to consult the compiler user manual to understand the size of long. It is typically 4 or 8 bytes.
- ✓ **unsigned long** is a integer data type to store unsigned integer data.

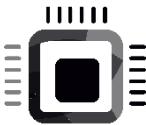


sizeof operator

- **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.



Variables



What are variables ?

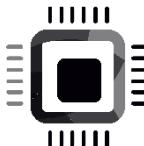
```
#include <stdio.h>

int main ()
{
    /* valueA and valueB are variables of type char */
    char valueA = 20;
    char valueB = 30;

    printf("Size = %d\n", sizeof( valueA ) );

    return 0;
}
```

- Variables are identifiers for your data.
- Data are usually stored in computer memory.
- So, a variable acts as a label to a memory location where the data is stored
- Variable names are not stored inside the computer memory, and the compiler replaces them with memory location addresses during data manipulation.



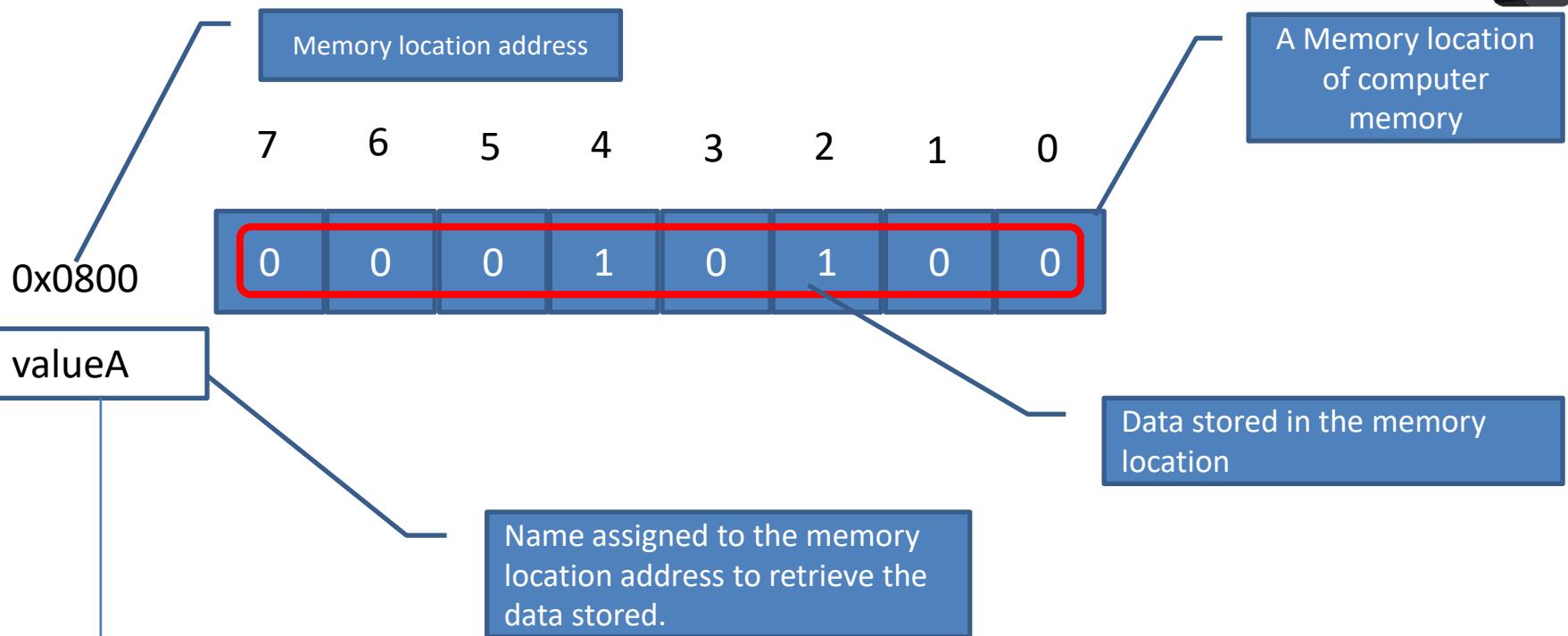
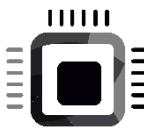
8 bits

0x0807	0	1	1	0	0	1	1	1
0x0806	1	1	0	0	0	1	0	0
0x0805	0	1	1	0	0	1	1	1
0x0804	1	1	1	1	0	1	1	1
0x0803	1	1	1	0	0	1	1	0
0x0802	0	1	1	0	0	1	1	0
0x0801	0	1	1	1	0	1	1	1
0x0800	0	1	1	1	0	1	1	1

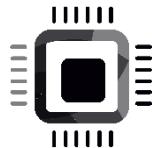
Memory location addresses
(pointers)

Value stored in a memory location 0x0803

Different
memory
locations
of the
computer
memory

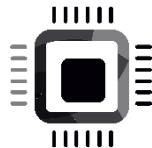


Variable name just exists for programming convenience and doesn't exist post-compilation, only its associated memory location address does.



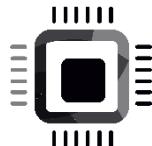
Defining Variables

- ✓ Before you use a variable, you have to define it.
- ✓ Variable definition (sometimes also called a variable declaration) is nothing more than letting the compiler know you will need some memory space for your program data so it can reserve some.



Defining Variables

- ✓ To define a variable, you only need to state its type, followed by a variable name.
- ✓ e.g., If the data type of the variable is **char**, then compiler reserves 1 byte. If the data type is **int** then compiler reserves 4 bytes .



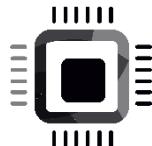
Syntax : Defining Variables

‘C’ Syntax of variable definition :

<data type> <variable_name> ;

Example :

char myExamScore; //This is called variable definition

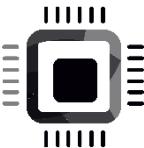


Variable initialization

```
char myExamScore ; //Variable definition
```

```
myExamScore = 25; //This is called variable initialization
```

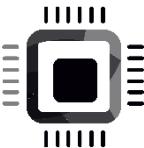
Observe that variable initialization is followed by first defining the variable .



This is illegal

```
myExamScore = 25; //variable initialization  
char myExamScore ; //Variable definition
```

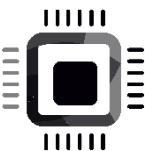
When the compiler compiles these statements, it sees that “myExamScore “ which is not defined before getting assigned by a value. So, the compiler throws an error here.



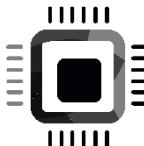
This is legal

- Variable definition and initialization can be done in one single statement. The below statement is legal from the compiler point of view.

char myExamScore = 25;

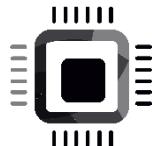


Rules for naming a variable



Rules for naming a variable

- Make sure that the variable name is not more than 30 characters. Some compilers may issue errors.
- A variable name can only contain alphabets(both uppercase and lowercase), digits and underscore.
- The first letter of the variable cannot be a digit. It should be either an alphabet or underscore
- You cannot use ‘C’ standard reserved keywords as variable names.



C99 reserved keywords

6.4.1 Keywords

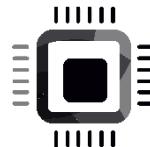
Syntax

- 1 *keyword*: one of

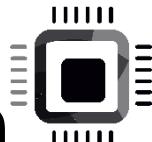
auto	enum	restrict	unsigned
break	extern	return	void
case	float	short	volatile
char	for	signed	while
const	goto	sizeof	_Bool
continue	if	static	_Complex
default	inline	struct	_Imaginary
do	int	switch	
double	long	typedef	
else	register	union	

Semantics

- 2 The above tokens (case sensitive) are reserved (in translation phases 7 and 8) for use as keywords, and shall not be used otherwise. The keyword **_Imaginary** is reserved for specifying imaginary types.⁵⁹⁾

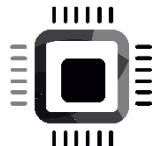


```
main.c
 9 #include <stdio.h>
10
11 int main()
12 {
13     char _11111111111111;           //legal
14
15     char 89_;                      //illegal
16
17     long long mySalary123_0;       //legal
18
19     int _____myAge;              //legal
20
21     char MYNAME;                 //legal
22
23     int %_temperatureValue;      //illegal
24
25     int my car number;          //illegal
26
27     int 1456;                    //illegal
28
```



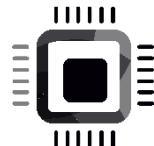
Variable definition vs. declaration

- A variable is **defined** when the compiler generates instructions to allocate the storage for the variable.
- A variable is **declared** when the compiler is informed that a variable exists along with its type. The compiler does not generate instructions to allocate the storage for the variable at that point.
- A variable definition is also a declaration, but not all variable declarations are definitions.



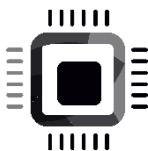
Variable scopes

- 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

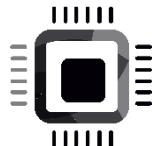


Variable scopes

- 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

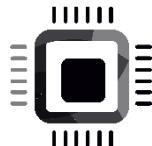


Summary global variables

Scope – Global variables are visible to all the functions of a program. They are everywhere. Even you can access global variables from another file of the project.

Default value – All uninitialized global variables will have 0 as default value.

Lifetime – Till the end of the execution of the program.

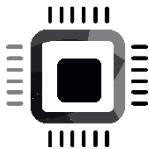


Summary local variables

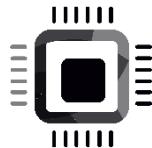
Scope – Within the body of the function. Local variables lose existence once the execution control comes out of the function body.

Default value – unpredictable (garbage value).

Lifetime – Till the end of the execution of a function in which a variable is defined.



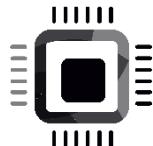
Storage classes in 'C'



The Storage Classes in ‘C’ Language decides:

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

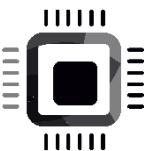
Scope, visibility, and lifetime are features of a variable . These features can be modified by using storage class specifiers.



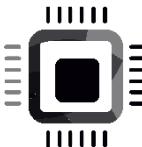
Storage class specifiers in ‘C’

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.



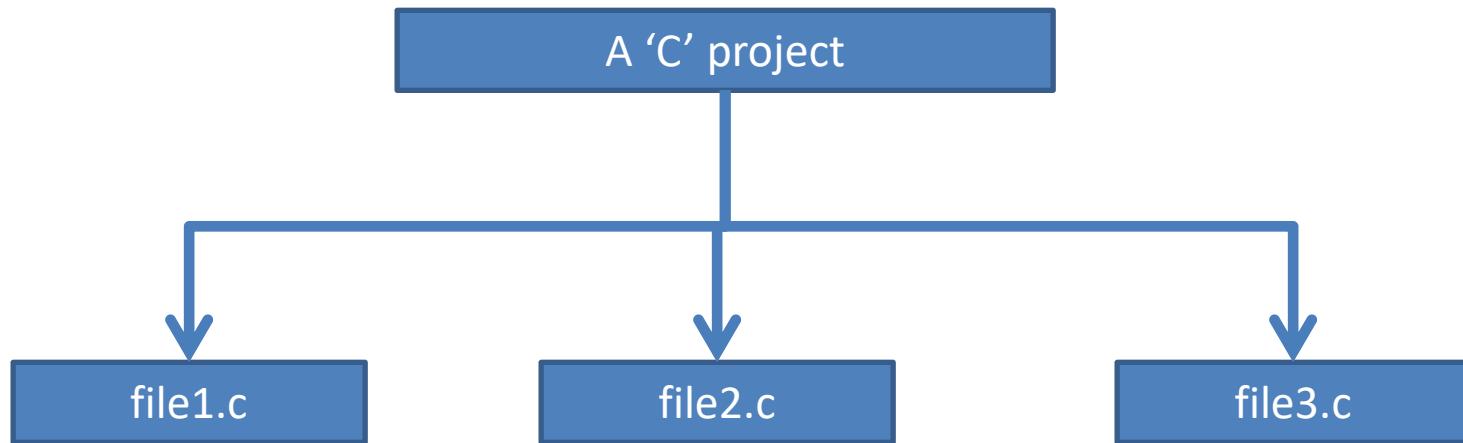
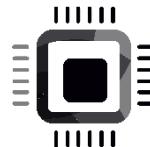
```
void fun1(void);
void fun2(void);

int main()
{
    printf("Hello World");
    return 0;
}

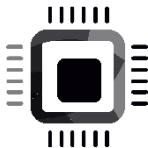
void fun1(void)
{
    static int fun1_var;
}

void fun2(void)
{
    int fun2_var;
```

Here this variable is treated as a global variable but private to fun1, and it never dies even if fun1 returns



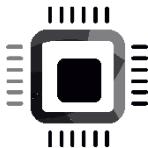
A 'C' project can be a collection of multiple source files, and a 'static**' storage class specifier can be used to manage the visibility of the variables across various files effectively.**



Extern

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

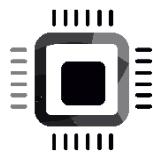
'extern' storage class specifier can also be used during the function call, when the function is defined outside the scope of the file.



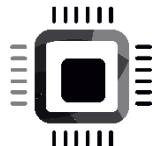
Extern

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.

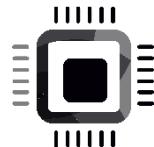


Address of a variable



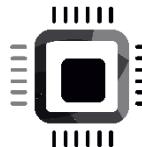
ASCII codes

- The American National Standards Institute (ANSI), which developed ANSI C, also developed the ASCII codes.
- ASCII stands for “American Standard Code for Information Interchange”

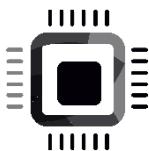


ASCII codes

- By using ASCII standard, you can encode 128 different characters.
- That's why to encode any ASCII character you just need 7 bits. (you can use char data type)

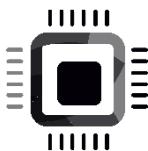


Dec	Hex	Oct	Chr	Dec	Hex	Oct	HTML	Chr	Dec	Hex	Oct	HTML	Chr	Dec	Hex	Oct	HTML	Chr
0	000	NULL		32	20	040	 	Space	64	40	100	@	@	96	60	140	`	`
1	001	Start of Header		33	21	041	!	!	65	41	101	A	A	97	61	141	a	a
2	002	Start of Text		34	22	042	"	"	66	42	102	B	B	98	62	142	b	b
3	003	End of Text		35	23	043	#	#	67	43	103	C	C	99	63	143	c	c
4	004	End of Transmission		36	24	044	$	\$	68	44	104	D	D	100	64	144	d	d
5	005	Enquiry		37	25	045	%	%	69	45	105	E	E	101	65	145	e	e
6	006	Acknowledgment		38	26	046	&	&	70	46	106	F	F	102	66	146	f	f
7	007	Bell		39	27	047	'	'	71	47	107	G	G	103	67	147	g	g
8	010	Backspace		40	28	050	((72	48	110	H	H	104	68	150	h	h
9	011	Horizontal Tab		41	29	051))	73	49	111	I	I	105	69	151	i	i
10	A	Line feed		42	2A	052	*	*	74	4A	112	J	J	106	6A	152	j	j
11	B	Vertical Tab		43	2B	053	+	+	75	4B	113	K	K	107	6B	153	k	k
12	C	Form feed		44	2C	054	,	,	76	4C	114	L	L	108	6C	154	l	l
13	D	Carriage return		45	2D	055	-	-	77	4D	115	M	M	109	6D	155	m	m
14	E	Shift Out		46	2E	056	.	.	78	4E	116	N	N	110	6E	156	n	n
15	F	Shift In		47	2F	057	/	/	79	4F	117	O	O	111	6F	157	o	o
16	10	Data Link Escape		48	30	060	0	0	80	50	120	P	P	112	70	160	p	p
17	11	Device Control 1		49	31	061	1	1	81	51	121	Q	Q	113	71	161	q	q
18	12	Device Control 2		50	32	062	2	2	82	52	122	R	R	114	72	162	r	r
19	13	Device Control 3		51	33	063	3	3	83	53	123	S	S	115	73	163	s	s
20	14	Device Control 4		52	34	064	4	4	84	54	124	T	T	116	74	164	t	t
21	15	Negative Ack.		53	35	065	5	5	85	55	125	U	U	117	75	165	u	u
22	16	Synchronous idle		54	36	066	6	6	86	56	126	V	V	118	76	166	v	v
23	17	End of Trans. Block		55	37	067	7	7	87	57	127	W	W	119	77	167	w	w
24	18	Cancel		56	38	070	8	8	88	58	130	X	X	120	78	170	x	x
25	19	End of Medium		57	39	071	9	9	89	59	131	Y	Y	121	79	171	y	y
26	1A	Substitute		58	3A	072	:	:	90	5A	132	Z	Z	122	7A	172	z	z
27	1B	Escape		59	3B	073	;	;	91	5B	133	[[123	7B	173	{	{
28	1C	File Separator		60	3C	074	<	<	92	5C	134	\	\	124	7C	174	|	
29	1D	Group Separator		61	3D	075	=	=	93	5D	135]]	125	7D	175	}	}
30	1E	Record Separator		62	3E	076	>	>	94	5E	136	^	^	126	7E	176	~	~
31	1F	Unit Separator		63	3F	077	?	?	95	5F	137	_	_	127	7F	177		Del



A p p l e :

For the machine everything is number.



65

112

112

108

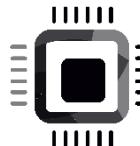
101

58

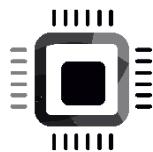
41



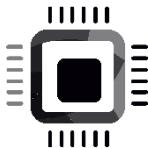
For the machine everything is number.



Dec	Hex	Oct	Chr	Dec	Hex	Oct	HTML	Chr	Dec	Hex	Oct	HTML	Chr	Dec	Hex	Oct	HTML	Chr
0	000	NULL		32	20	040	 	Space	64	40	100	@	@	96	60	140	`	'
1	001	Start of Header		33	21	041	!	!	65	41	101	A	A	97	61	141	a	a
2	002	Start of Text		34	22	042	"	"	66	42	102	B	B	98	62	142	b	b
3	003	End of Text		35	23	043	#	#	67	43	103	C	C	99	63	143	c	c
4	004	End of Transmission		36	24	044	$	\$	68	44	104	D	D	100	64	144	d	d
5	005	Enquiry		37	25	045	%	%	69	45	105	E	E	101	65	145	e	e
6	006	Acknowledgment		38	26	046	&	&	70	46	106	F	F	102	66	146	f	f
7	007	Bell		39	27	047	'	'	71	47	107	G	G	103	67	147	g	g
8	010	Backspace		40	28	050	((72	48	110	H	H	104	68	150	h	h
9	011	Horizontal Tab		41	29	051))	73	49	111	I	I	105	69	151	i	i
10	A	Line feed		42	2A	052	*	*	74	4A	112	J	J	106	6A	152	j	j
11	B	Vertical Tab		43	2B	053	+	+	75	4B	113	K	K	107	6B	153	k	k
12	C	Form feed		44	2C	054	,	,	76	4C	114	L	L	108	6C	154	l	l
13	D	Carriage return		45	2D	055	-	-	77	4D	115	M	M	109	6D	155	m	m
14	E	Shift Out		46	2E	056	.	.	78	4E	116	N	N	110	6E	156	n	n
15	F	Shift In		47	2F	057	/	/	79	4F	117	O	O	111	6F	157	o	o
16	10	Data Link Escape		48	30	060	0	0	80	50	120	P	P	112	70	160	p	p
17	11	Device Control 1		49	31	061	1	1	81	51	121	Q	Q	113	71	161	q	q
18	12	Device Control 2		50	32	062	2	2	82	52	122	R	R	114	72	162	r	r
19	13	Device Control 3		51	33	063	3	3	83	53	123	S	S	115	73	163	s	s
20	14	Device Control 4		52	34	064	4	4	84	54	124	T	T	116	74	164	t	t
21	15	Negative Ack.		53	35	065	5	5	85	55	125	U	U	117	75	165	u	u
22	16	Synchronous idle		54	36	066	6	6	86	56	126	V	V	118	76	166	v	v
23	17	End of Trans. Block		55	37	067	7	7	87	57	127	W	W	119	77	167	w	w
24	18	Cancel		56	38	070	8	8	88	58	130	X	X	120	78	170	x	x
25	19	End of Medium		57	39	071	9	9	89	59	131	Y	Y	121	79	171	y	y
26	1A	Substitute		58	3A	072	:	:	90	5A	132	Z	Z	122	7A	172	z	z
27	1B	Escape		59	3B	073	;	;	91	5B	133	[[123	7B	173	{	{
28	1C	File Separator		60	3C	074	<	<	92	5C	134	\	\	124	7C	174	|	
29	1D	Group Separator		61	3D	075	=	=	93	5D	135]]	125	7D	175	}	}
30	1E	Record Separator		62	3E	076	>	>	94	5E	136	^	^	126	7E	176	~	~
31	1F	Unit Separator		63	3F	077	?	?	95	5F	137	_	_	127	7F	177		Del



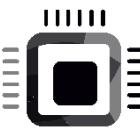
Address of a variable



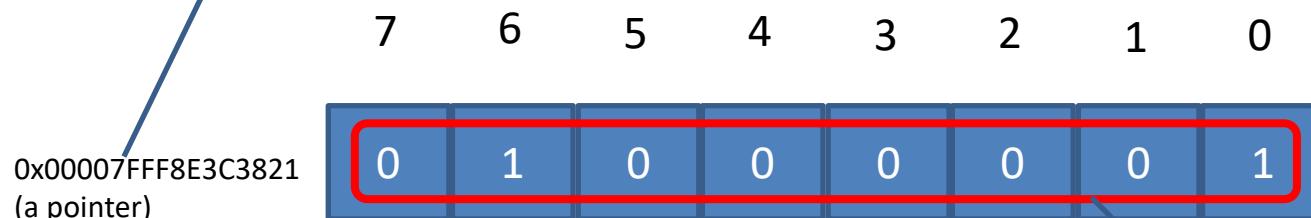
Write a program to print the address of variables.

int myData;

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



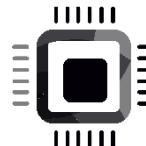
Memory location address.
Address size is 8 bytes(64 bit machine)



a1

Variable name

Data stored in the memory location (ASCII code of 'A')



This represent a variable.
Variables are represented by
variable data type

TYPE : unsigned long int

unsigned long int addressOfa1 =

This is a pointer data.
Pointer data are
represented by pointer data
type in 'C'

TYPE : char *

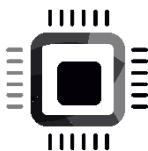
&a1;

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)

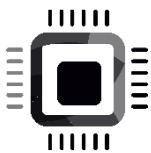
This is not just a
number.
This is a pointer



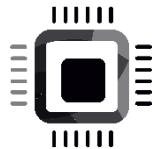
```
unsigned long int addressOfa1 = (unsigned long int) &a1;
```

Type casting

Now, this is a number of type unsigned long int.
Not a pointer

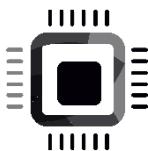


scanf



scanf

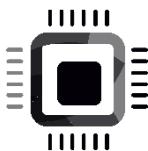
- scanf is a standard library function that allows you to read input from standard in.
- Standard in for us is generally the keyboard
- By using scanf library function you can read both characters and numbers from the keyboard.



```
int age ;  
printf("Enter your age : ");  
scanf("%d", &age);
```

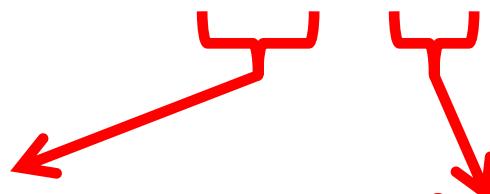
scanf reads an integer(a number)
which the user enters

scanf puts that read value
“At the address of” ‘age’ variable

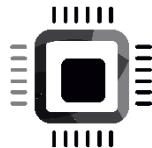


```
int c;  
printf("Enter a character: ");  
scanf("%c", &c);
```

scanf reads a character
which the user enters

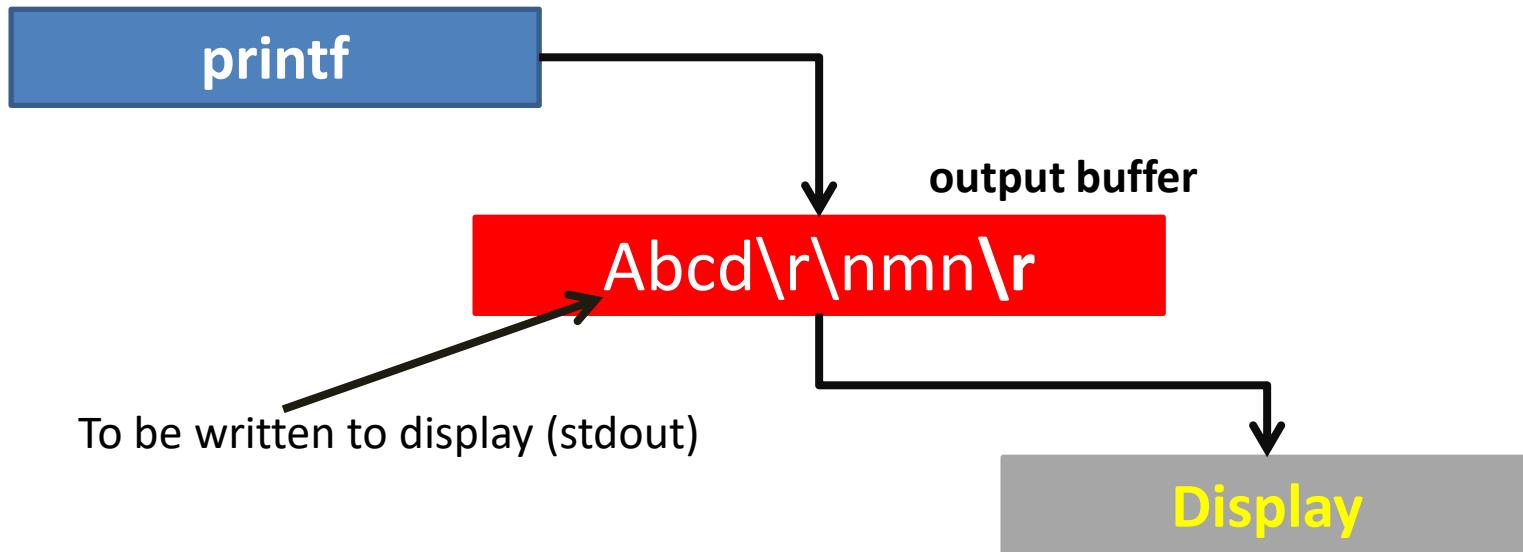
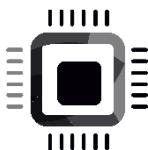


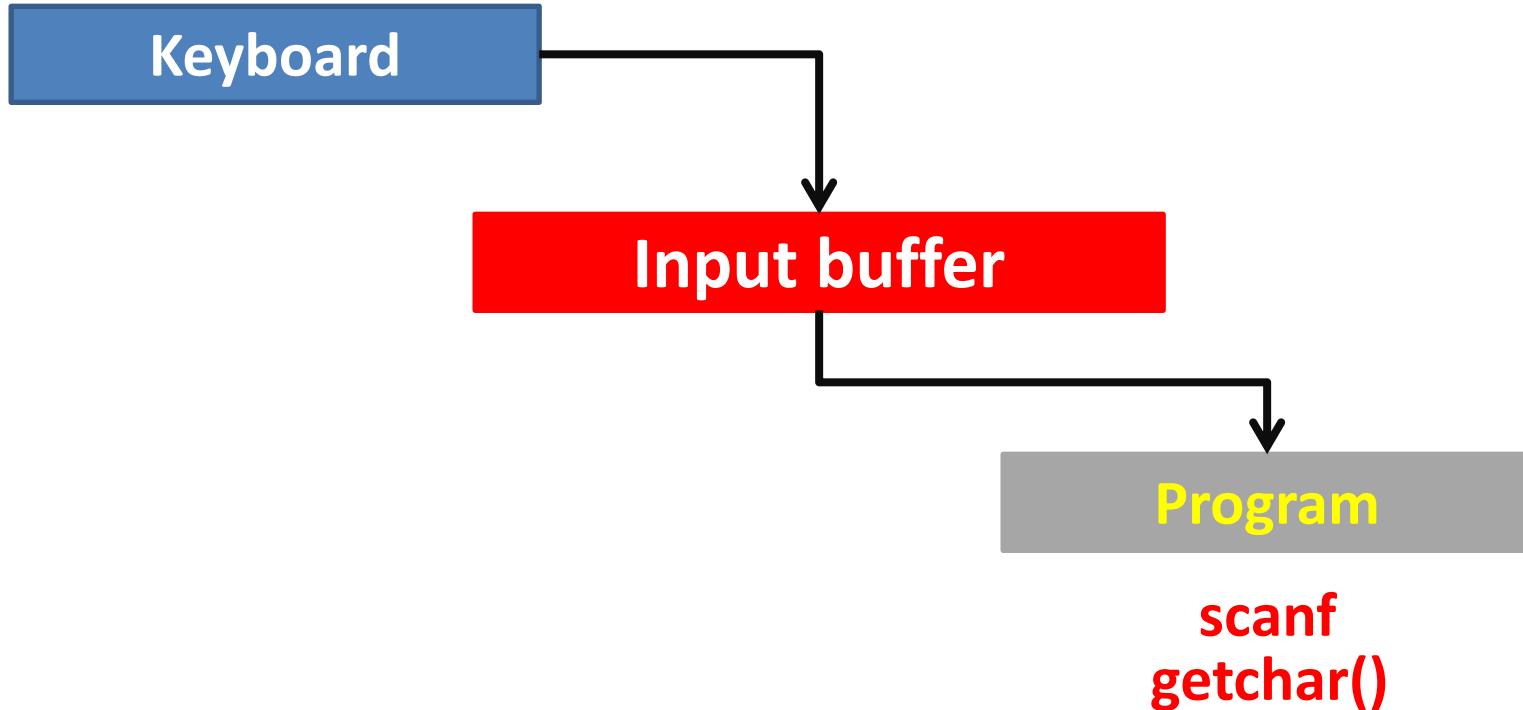
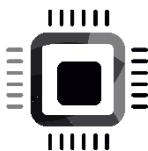
scanf puts that read value
“At the address of” ‘c’ variable

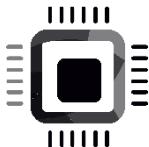


getchar()

- If you just want to read a single character from the keyboard in ASCII format then just use `getchar()`.
- `getchar()` function takes no argument and just returns an int value which happens to be the ASCII value of the key pressed
- `int a = getchar(); // Here the program hangs until you press a key followed by pressing the enter key.`







10

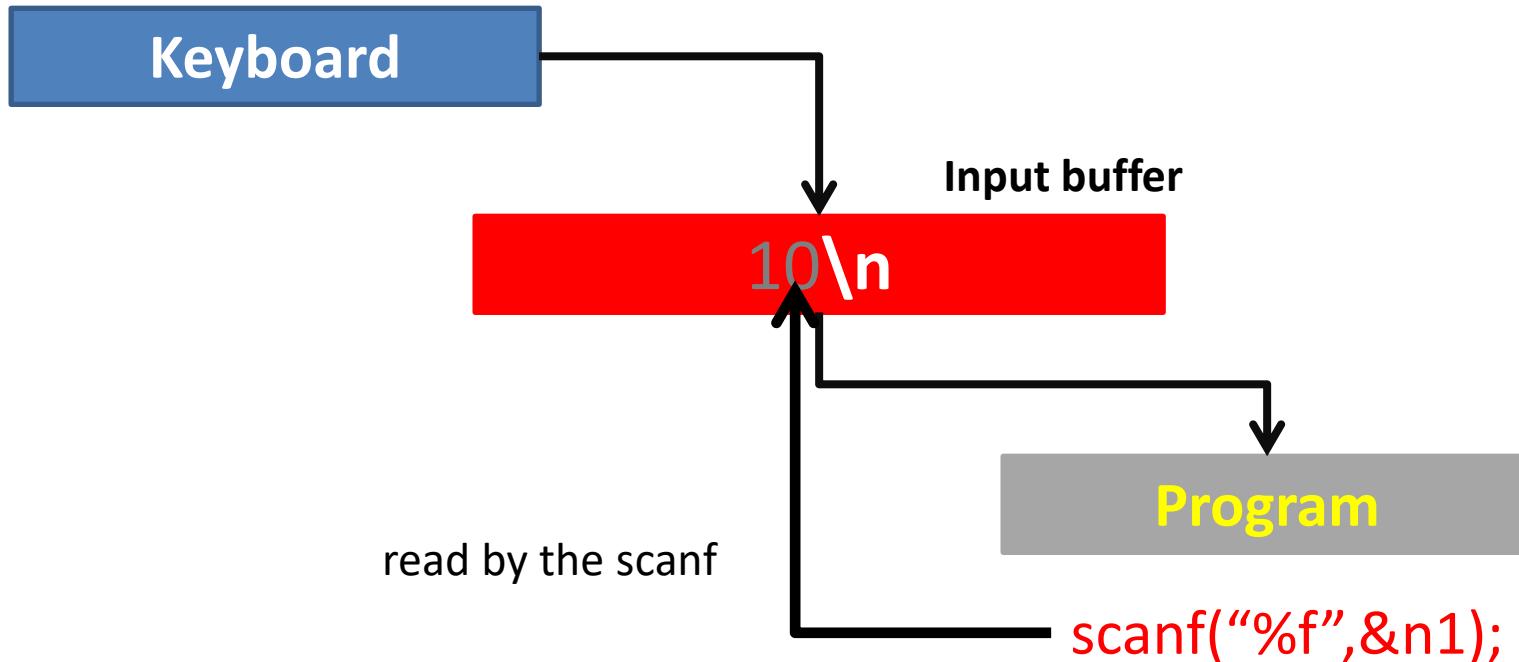
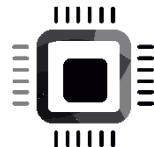
← Enter (\n) New line

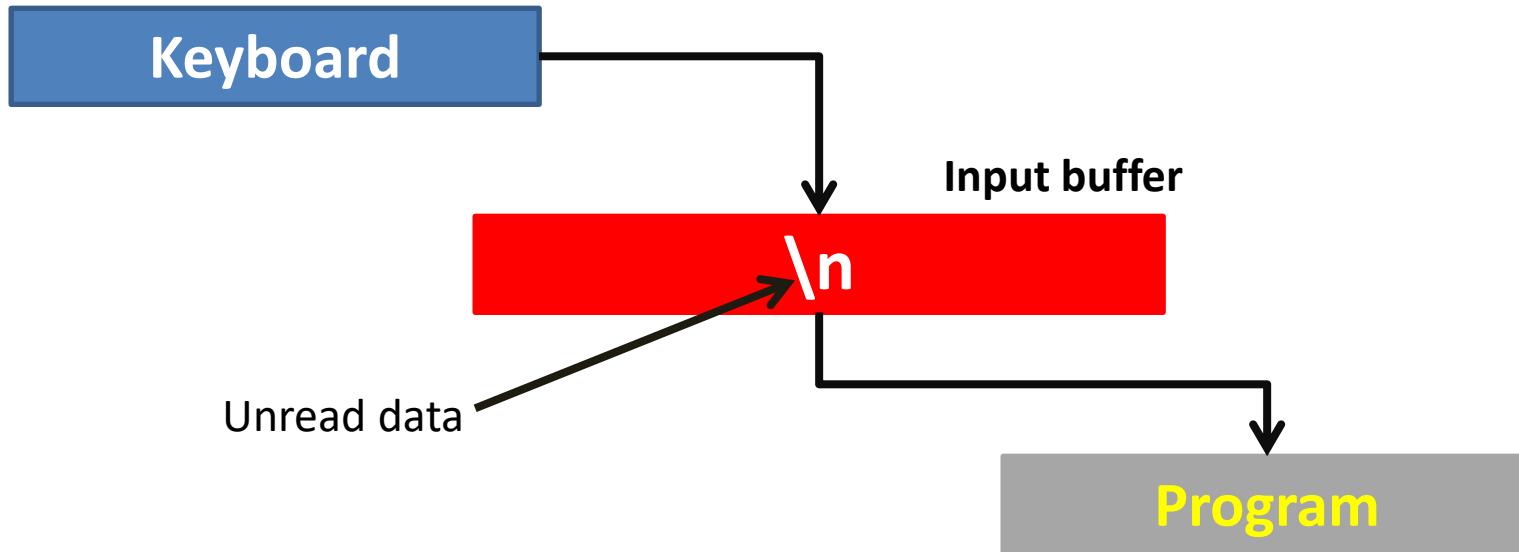
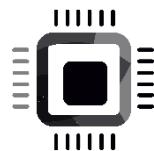
Keyboard

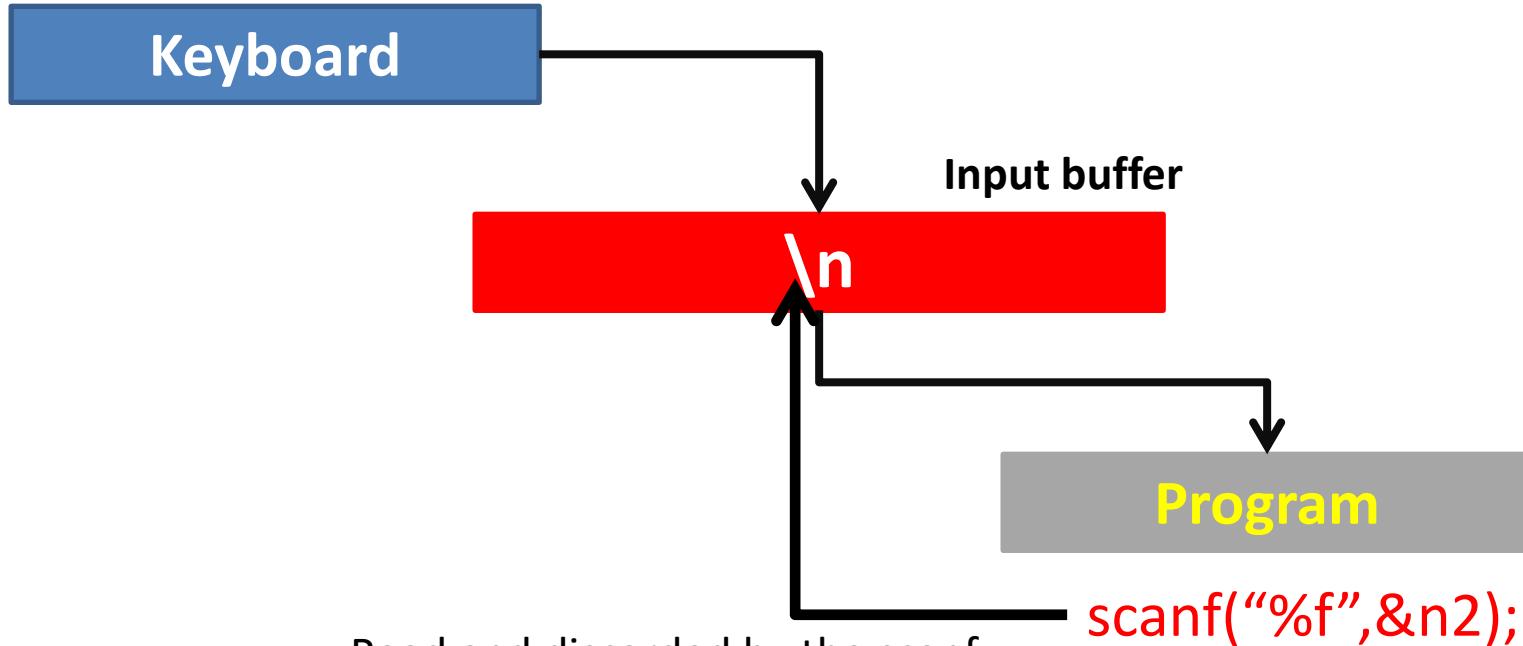
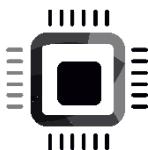
Input buffer

Program

`scanf("%f",&n1);`

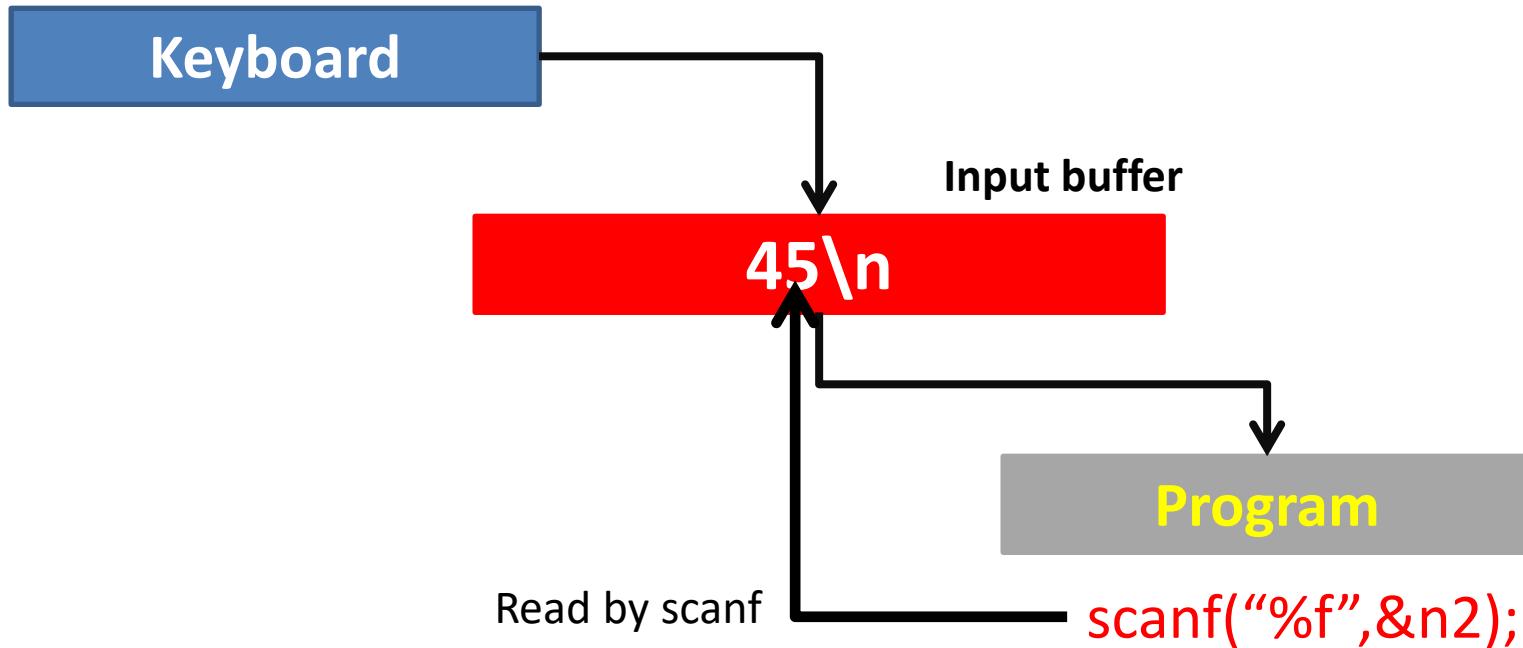
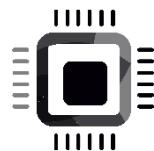


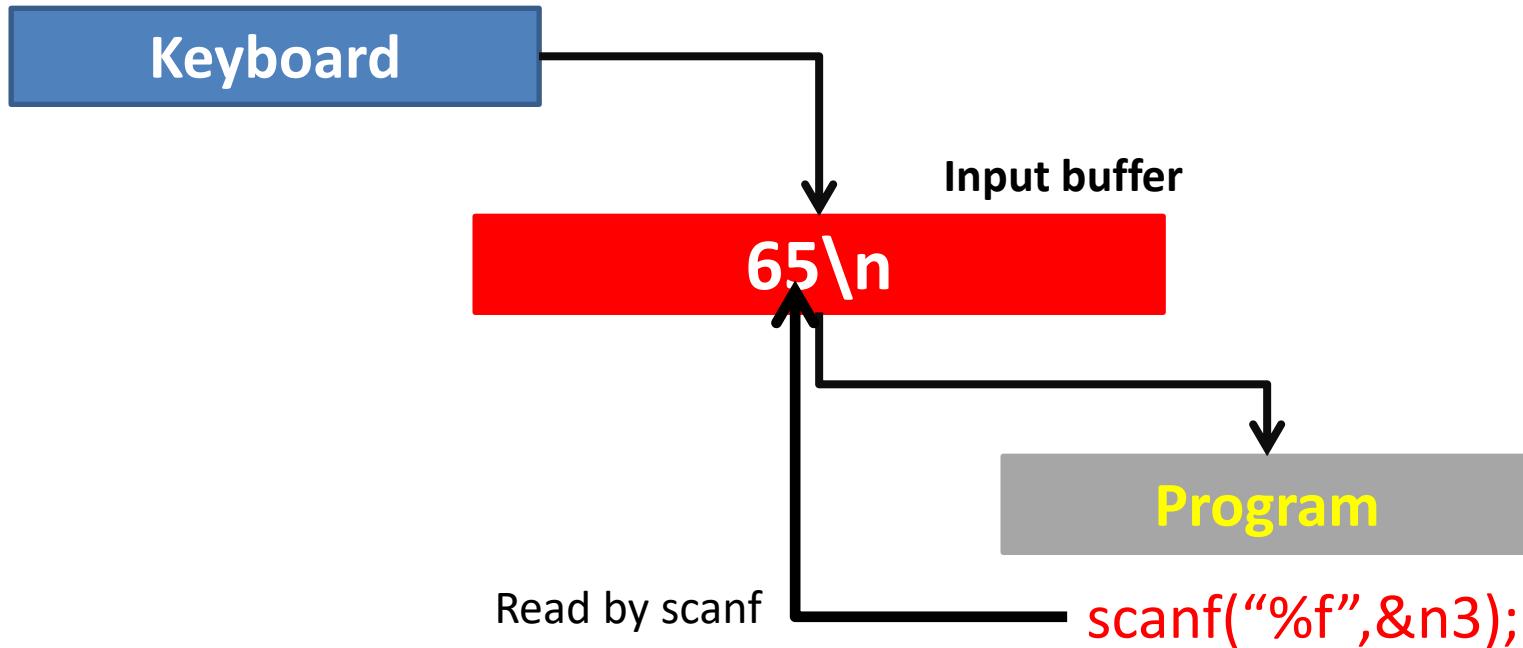
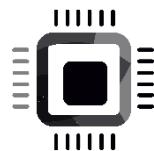


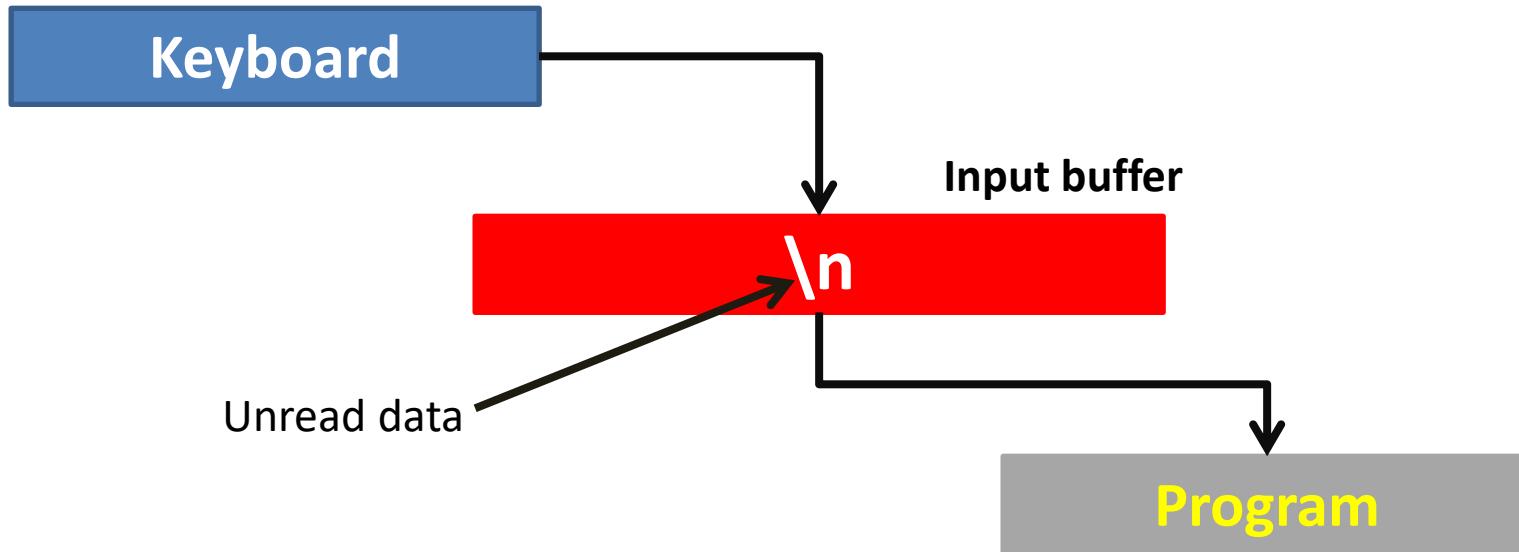
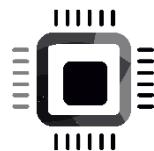


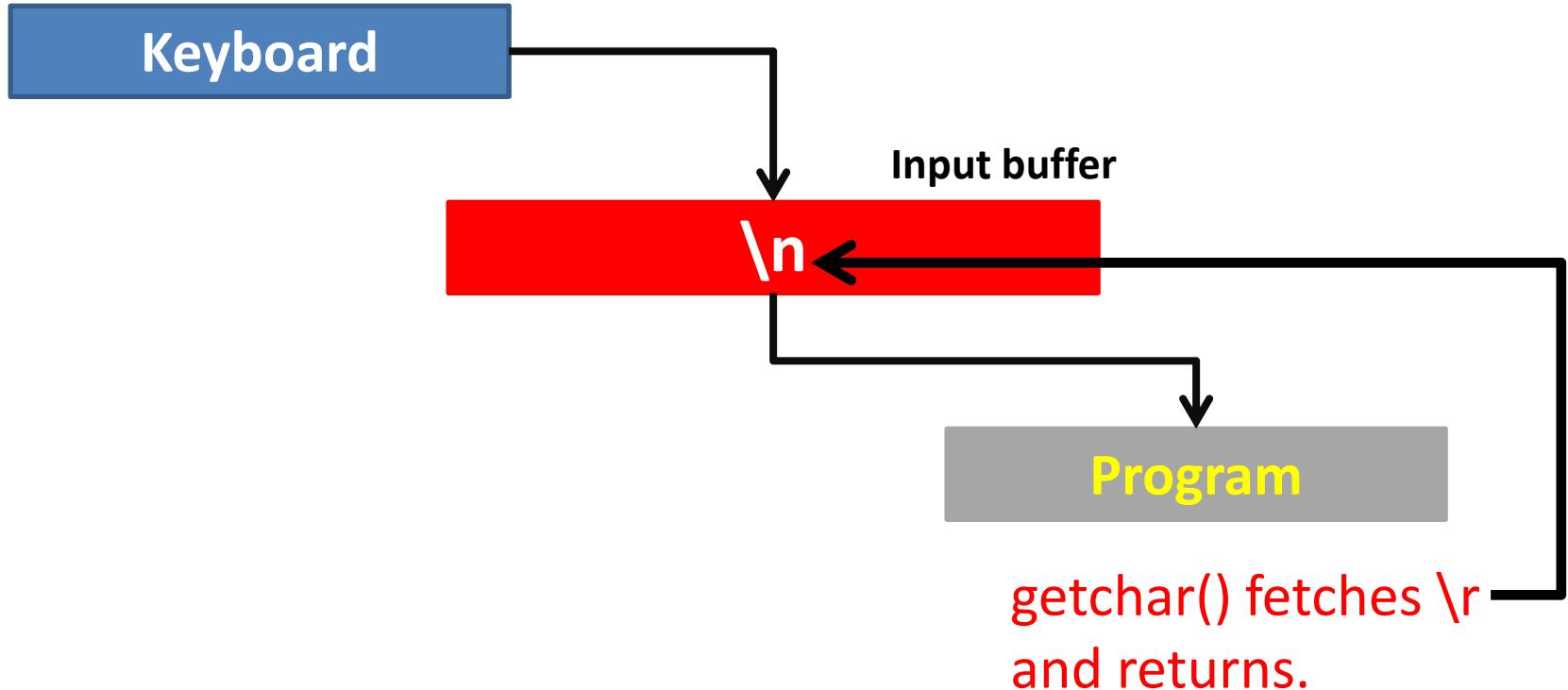
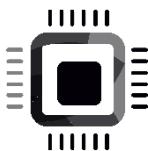
Read and discarded by the `scanf`
because its not a valid number
but a special character . `Scanf` waits

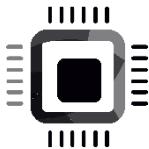
`scanf("%f", &n2);`





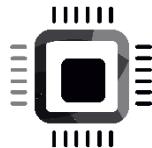






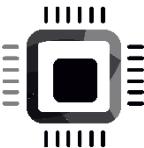
Exercise

Write a program which takes 3 numbers from the user ,
computes and prints the average of those numbers.



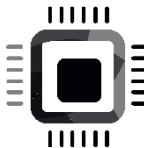
Exercise

Write a program which takes 6 characters (alphabets, numbers and special characters) from the user and prints the **ASCII** codes of the entered characters.

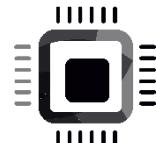


Functions in ‘C’

- 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.



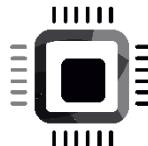
- In your program, if you want to perform the same set of tasks again and again, then create a function to do that task and call it every time you need to perform that task.
- This reduces code redundancy and brings modularity to your program and also decreases the final executable size of your program.



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.

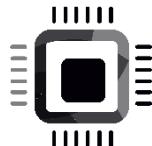


```
int main()
{
    /* ... */
}
```

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.

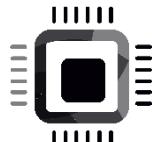


'main' takes only zero or two arguments

main() is the special function in C from where execution of a program starts and ends.

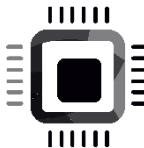
main() function returns the status of a program to the Parent process, showing the success or failure of the program. 0 means SUCCESS. Non zero ERROR.

main() function suppose to return an *int value* to the calling process as per the standard (C89 and above)



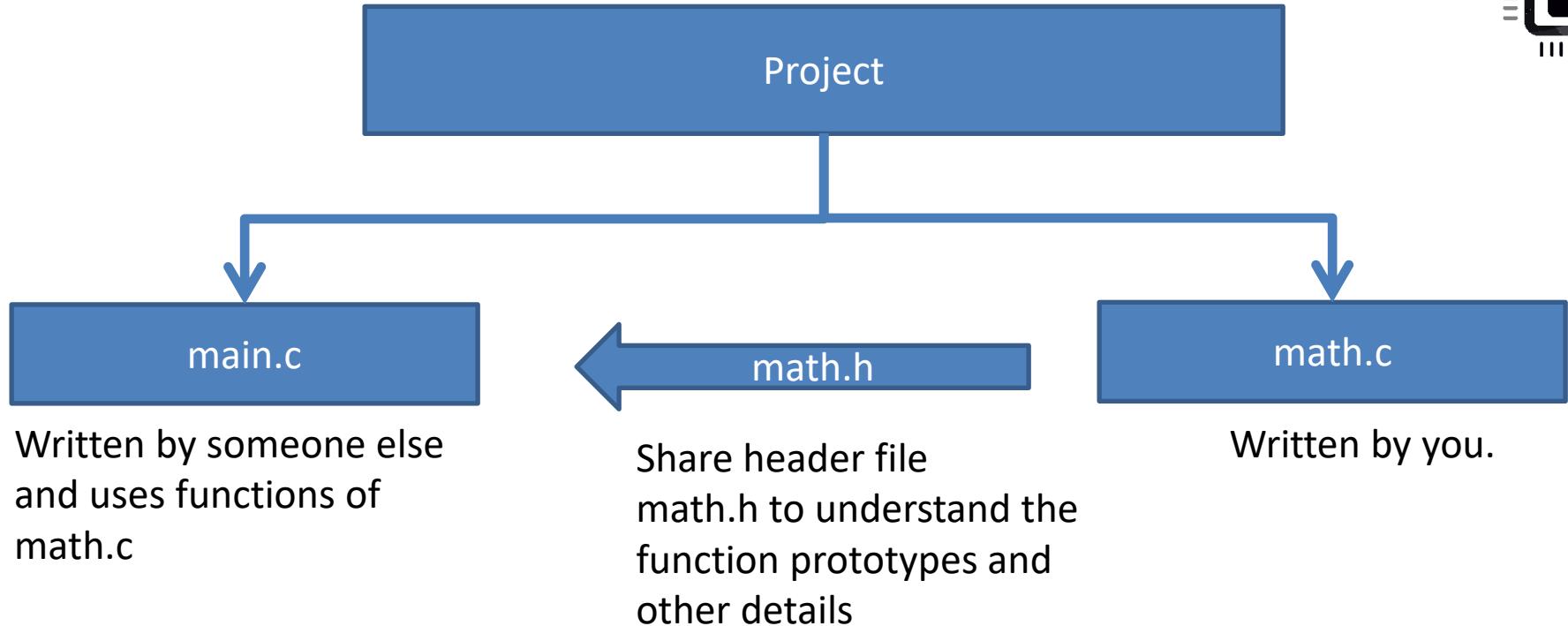
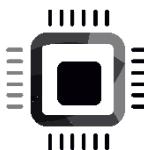
Function prototype (declaration)

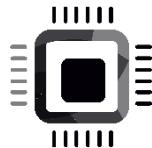
- In ‘C’ functions first have to be declared before they are used.
- Prototype lets compiler to know about the return data type, argument list and their data type and order of arguments being passed to the function .



Write a program to do mathematical operations such as addition, subtraction, multiplication, and division of integers.

- 1)Do all the mathematical operations in separate functions in a file called math.c
- 2)Call the functions from the main.c to test them.

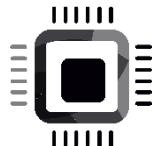




Type casting

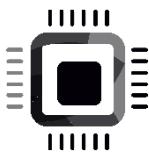
Typecasting is a way of converting a variable or data from one data type to another data type.

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

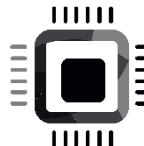


There are 2 types of casting

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



Pointers



Width 8 bits

0x00007FFF8E3C3828
0x00007FFF8E3C3827
0x00007FFF8E3C3826
0x00007FFF8E3C3825
0x00007FFF8E3C3824
0x00007FFF8E3C3823
0x00007FFF8E3C3822
0x00007FFF8E3C3821

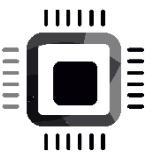
0	1	1	0	0	1	1	1
1	1	0	0	0	1	0	0
0	1	1	0	0	1	1	1
1	1	1	1	0	1	1	1
1	1	1	0	0	1	1	0
0	1	1	0	0	1	1	0
0	1	1	1	0	1	1	1
0	1	1	1	0	1	1	1

Different
memory
locations of
the computer
memory

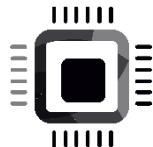
Memory location addresses (pointers)

Value stored in a memory location
0x00007FFF8E3C3824

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

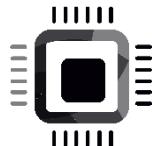


Now, let's see how to store a pointer inside the program.



Pointer variable definition

<pointer data type > <Variable Name > ;



Pointer data types

char*

int*

long long int*

float*

double*

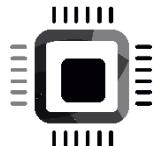
unsigned char*

unsigned int*

unsigned long long int*

unsigned float*

unsigned double*

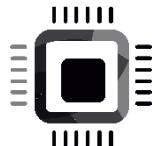


Pointer variable definition and initialization

This is a pointer variable
of type **char***



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

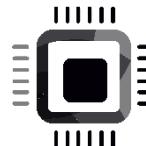


Pointer variable definition and initialization

This is a pointer variable
of type **char***

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

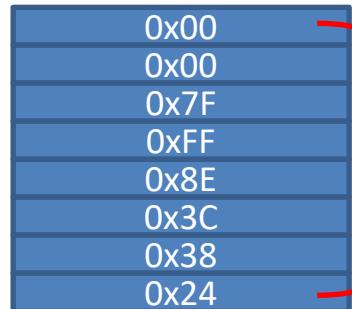
Compiler allocates 8 bytes for this variable to store this pointer.



0x1F007FFF8E3C4828
0x1F007FFF8E3C4827
0x1F007FFF8E3C4826
0x1F007FFF8E3C4825
0x1F007FFF8E3C4824
0x1F007FFF8E3C4823
0x1F007FFF8E3C4822
0x1F007FFF8E3C4821

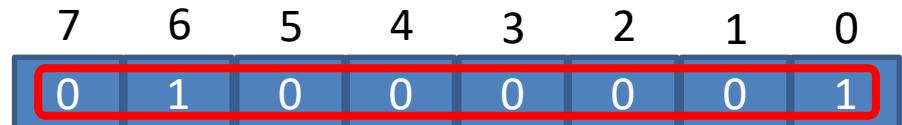
address1

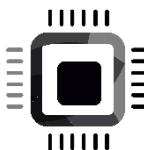
Address of the pointer variable



Pointer variable creation and initialization

0x00007FFF8E3C3824





0x00007FFF8E3C3824

0x1F007FFF8E3C4821

address1

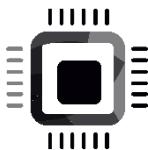
An initialized pointer variable



0x85

0x00007FFF8E3C3824

A memory location address and value



char* address1

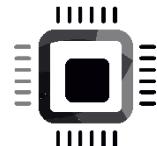
int* address1

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.

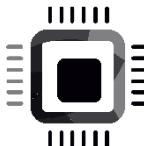


Pointer variable definition and initialization

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

?

char* address1 = (char*) 0x00007FFF8E3C3824;



Pointer variable definition and initialization

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

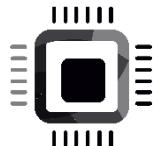
?

`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



Pointer variable definition and initialization

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

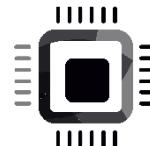
?

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



The pointer data type decides the behavior of the operations carried out on the pointer variable.

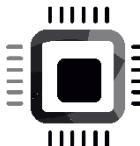
Operations : read, write, increment, decrement



```
/* 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

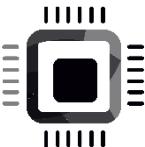
```
/* Read operation on address1 variable yields 1 byte of data */  
char* address1 = (char*) 0x00007FFF8E3C3824;
```

Pointer variable to int type data

```
/* 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

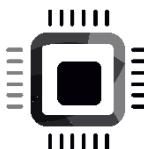
```
char data = *address1; //dereferencing a pointer to read data
```



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

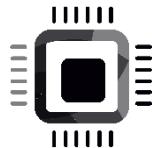
```
char data = *address1;  
(data = 0x85)
```



0x85

0x00007FFF8E3C3824

A memory location address and value

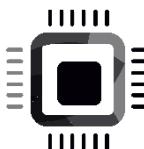


Write operation on the pointer

```
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

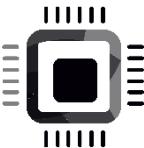
0x85
0x00007FFF8E3C3824
A memory location address and value

`*address1 = 0x89;`

0x00007FFF8E3C3824
0x1F007FFF8E3C4821
address1
An initialized pointer variable

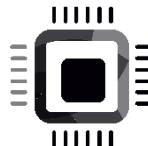
After

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



100

1

000000000061FE46 (Address of variable “data”)

data

char* pAddress = &data;

(Address of the variable “data” is stored in the pointer variable “pAddress ”)

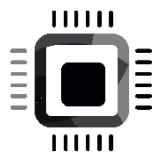
2

i.e pAddress = 000000000061FE46 ;

value = *pAddress;

Dereferencing the address 000000000061FE46 to read the value stored
100 will be stored in the variable “value”

3



***pAddress = 65;**

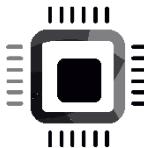
Dereferencing the address 00000000061FE46 to write the value 65
65 will be stored in the memory location addressed by 00000000061FE46

4

65

00000000061FE46 (Address of variable “data”)

data



char data = 100;

Here “data” is of type “char”

Therefore, “&data” will be of type “char* ”

int temp = 90;

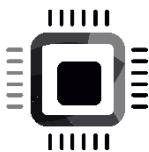
Here “temp ” is of type “int”

Therefore, “&temp” will be of type “int* ”

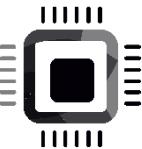
long long distance= 9000000;

Here “distance” is of type “long long int”

Therefore, “& distance” will be of type “long long int * ”



Effect of using different pointer data types on pointer operations .



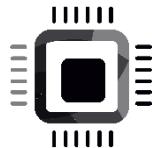
char *pAddress;

0xFF	0X0000000000403017	
0xFE	0X0000000000403016	
0xAB	0X0000000000403015	
0xCD	0X0000000000403014	
0x11	0X0000000000403013	
0x11	0X0000000000403012	
0x23	0X0000000000403011	
0x 45	0X0000000000403010 <i>(&g_data)</i>	pAddress

g_data

char *pAddress;

pAddress = pAddress + 1;



0xFF	0X0000000000403017
0xFE	0X0000000000403016
0xAB	0X0000000000403015
0xCD	0X0000000000403014
0x11	0X0000000000403013
0x11	0X0000000000403012
0x23	0X0000000000403011
0x 45	0X0000000000403010

g_data

0X0000000000403017

0X0000000000403016

0X0000000000403015

0X0000000000403014

0X0000000000403013

0X0000000000403012

0X0000000000403011

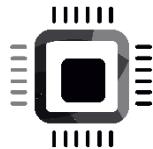
0X0000000000403010

pAddress

+ 1

char *pAddress;

pAddress = pAddress + 5;



0xFF
0xFE
0xAB
0xCD
0x11
0x11
0x23
0x 45

g_data

0X0000000000403017

0X0000000000403016

0X0000000000403015

0X0000000000403014

0X0000000000403013

0X0000000000403012

0X0000000000403011

0X0000000000403010

pAddress

+ 5 -

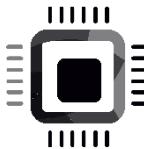
+ 4

+ 3

+ 2

+ 1

int *pAddress;



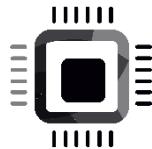
0xFF	0X0000000000403017
0xFE	0X0000000000403016
0xAB	0X0000000000403015
0xCD	0X0000000000403014
0x11	0X0000000000403013
0x11	0X0000000000403012
0x23	0X0000000000403011
0x 45	0X0000000000403010

g_data

pAddress

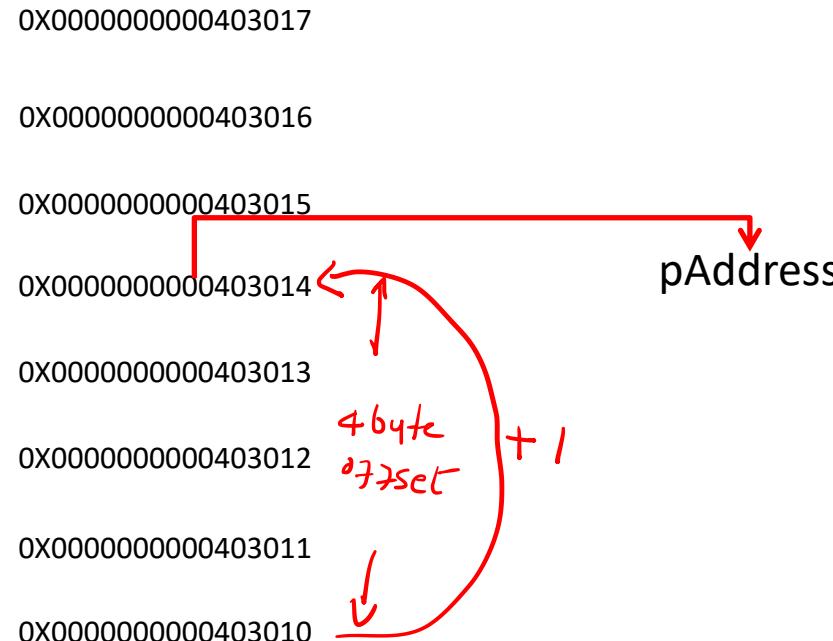
Int *pAddress;

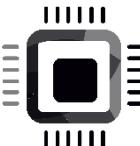
pAddress = pAddress + 1;



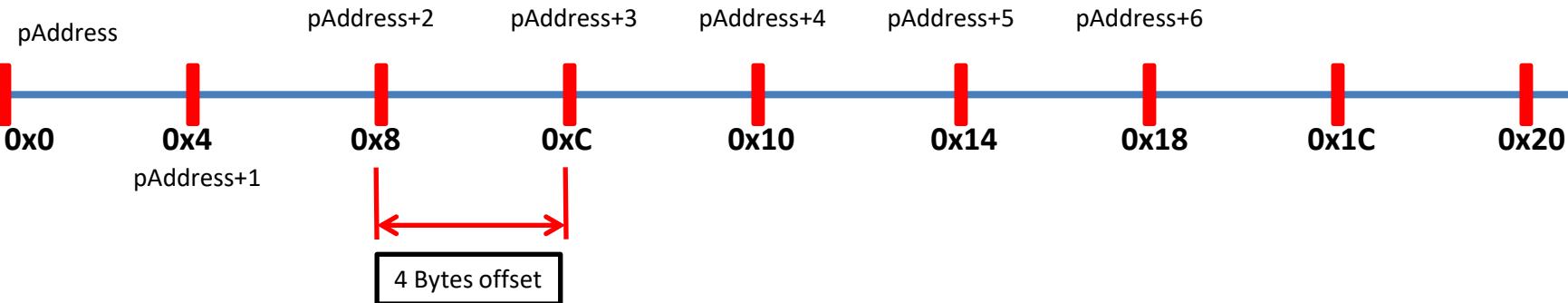
0xFF	0X000000000000403017
0xFE	0X000000000000403016
0xAB	0X000000000000403015
0xCD	0X000000000000403014
0x11	0X000000000000403013
0x11	0X000000000000403012
0x23	0X000000000000403011
0x 45	0X000000000000403010

g_data

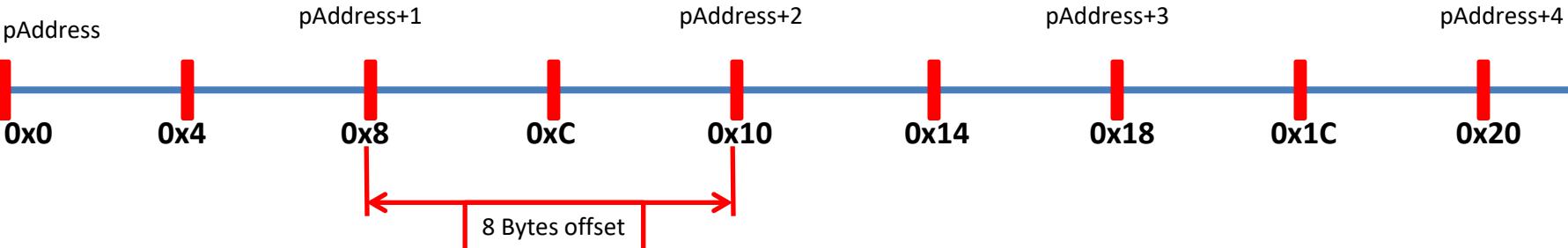


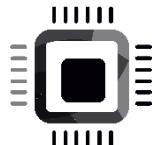


Int *pAddress;



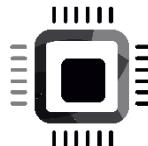
long long *pAddress;





How is Pointer used in embedded programming?

- Store data into required SRAM locations.
- For copying data from Peripheral register to SRAM memory and vice versa.
- To configure the peripheral registers. Because peripheral registers are memory-mapped and each register will be given unique address in the MCU memory map.
- Pointers to ISRs are stored in a vector table to handle the interrupts.
- Pointers are also used to configure the memory-mapped processor specific registers like interrupt configuration registers.



Importance of <stdint.h>

4.4.2 Integer Data Types

The MPLAB XC8 compiler supports integer data types with 1, 2, 3 and 4 byte sizes as well as a single bit type. **Table 4-3** shows the data types and their corresponding size and arithmetic type. The default type for each type is underlined.

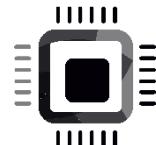
TABLE 4-3: INTEGER DATA TYPES

Type	Size (bits)	Arithmetic Type
<u>_bit</u>	1	Unsigned integer
<u>signed char</u>	8	Signed integer
<u>unsigned char</u>	8	Unsigned integer
<u>signed short</u>	16	Signed integer
<u>unsigned short</u>	16	Unsigned integer
<u>signed int</u>	16	Signed integer
<u>unsigned int</u>	16	Unsigned integer
<u>_int24</u>	24	Signed integer
<u>_uint24</u>	24	Unsigned integer
<u>signed long</u>	32	Signed integer
<u>unsigned long</u>	32	Unsigned integer
<u>signed long long</u>	32/64	Signed integer
<u>unsigned long long</u>	32/64	Unsigned integer

Source : MPLAB compiler guide

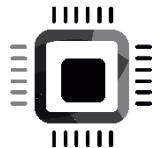
Type	Size in bits	Natural alignment in bytes	Range of values
<u>char</u>	8	1 (byte-aligned)	0 to 255 (unsigned) by default. -128 to 127 (signed) when compiled with <code>--signed_chars</code> .
<u>signed char</u>	8	1 (byte-aligned)	-128 to 127
<u>unsigned char</u>	8	1 (byte-aligned)	0 to 255
<u>(signed)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

The size of the data types depends upon the compiler.

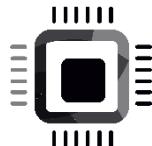


Importance of <stdint.h>

Each architecture(x86, 8051, PIC, ARM,etc.) has a natural, most-efficient size, and the designers, specifically compiler implementers use the natural native data size for speed and code size efficiency.

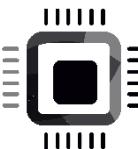


Portability Issues in C



Portability issues in “C” programming code due to “size of data types”

- ✓ In ‘C’ programming language the most commonly used data types “int” and “long” cause portability issues.
- ✓ The reason is that the storage size for ‘int’, ‘long’ type variable is not defined within the C standard (C90 or C99).
- ✓ The compiler vendors have the choice to define the storage size for the variable depending solely on hardware capabilities of the target platform, with respect to the minimum widths defined by the standard.



4.4.2 Integer Data Types

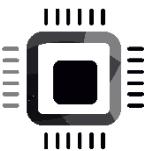
The MPLAB XC8 compiler supports integer data types with 1, 2, 3 and 4 byte sizes as well as a single bit type. Table 4-3 shows the data types and their corresponding size and arithmetic type. The default type for each type is underlined.

TABLE 4-3: INTEGER DATA TYPES

Type	Size (bits)	Arithmetic Type
<u>_bit</u>	1	Unsigned integer
<u>signed char</u>	8	Signed integer
<u>unsigned char</u>	8	Unsigned integer
<u>signed short</u>	16	Signed integer
<u>unsigned short</u>	16	Unsigned integer
<u>signed int</u>	16	Signed integer
<u>unsigned int</u>	16	Unsigned integer
<u>_int24</u>	24	Signed integer
<u>_uint24</u>	24	Unsigned integer
<u>signed long</u>	32	Signed integer
<u>unsigned long</u>	32	Unsigned integer
<u>signed long long</u>	32/64	Signed integer
<u>unsigned long long</u>	32/64	Unsigned integer

According to the compiler designer keeping the size of the “int” type variable as 2 bytes will be most efficient for data manipulation considering the underlying architecture of the PIC 8-bit microcontrollers

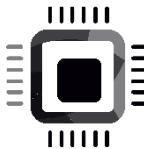
Why “int” is 2 bytes ?



Never use this

~~unsigned int count=0;
count++;~~

In order to get rid of portability issues due to data type size mismatch across different compilers , you have to stop using standard data type names,
instead use the data type alias names given by the header file stdint.h



stdint.h

- ✓ The standard library header file stdint.h defines fixed-width integers using alias data types for the standard data types available in ‘C’
- ✓ A fixed-width integer data type is an aliased data type that is based on the exact number of bits required to store the data



stdint.h helps you to choose an exact size for your variable and makes code portable no matter which compiler the code may be compiled on.

*stdint.h
aliased data
types*

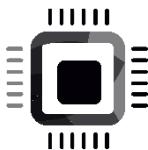
Exact Alias	Description	range
int8_t	excatly 8 bits signed	-128 to 127
uint8_t	excatly 8 bits unsigned	0 to 255
int16_t	excatly 16 bits signed	-32,768 to 32,767
uint16_t	excatly 16 bits unsigned	0 to 65,535
int32_t	excatly 32 bits signed	-2,147,483,648 to 2,147,483,647
uint32_t	excatly 32 bits unsigned	0 to 4,294,967,295
int64_t	excatly 64 bits signed	-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
uint64_t	excatly 64 bits unsigned	0 to 18,446,744,073,709,551,615



stdint.h helps you to choose an exact size for your variable and makes code portable no matter which compiler the code may be compiled on.

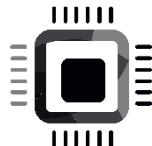
*stdint.h
aliased data
types*

Exact Alias		
int8_t		
uint8_t		
int16_t		
uint16_t		
int32_t		
uint32_t		
int64_t	exactly 64 bits signed	-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
uint64_t	exactly 64 bits unsigned	0 to 18,446,744,073,709,551,615



```
✓ uint32_t count=0;  
count++;  
If( count > 65,53  
{  
    //Do this task  
}
```

Here, doesn't matter under which compiler this code compiles, the compiler will always reserve 32bits for the variable by using suitable standard data type.

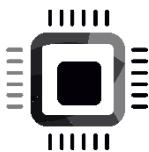


Some useful `stdint.h` aliases while programming

`uintmax_t` : defines the largest fixed-width unsigned integer possible on the system

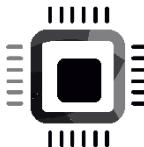
`intmax_t` : defines the largest fixed-width signed integer possible on the system

`uintptr_t` : defines a unsigned integer type that is wide enough to store the value of a pointer.



Operators in ‘C’

Operators in ‘C’



Unary operators

Operator	Type
<code>++, --</code>	Increment and decrement operator (Unary operator)
<code>+, -, *, /, %</code>	Arithmetic operators
<code><, <=, >, >=, ==, !=</code>	Relational operators
<code>&&, , !</code>	Logical Operators
<code>&, , <<, >>, ~, ^</code>	Bitwise operators
<code>=, +=, -=, *=, /=, %=</code>	Assignment operators
<code>?:</code>	Conditional operators

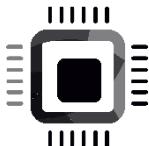
Binary operators



Ternary operators



An operator is a symbol that tells the compiler to perform a certain mathematical or logical manipulation on the operands



Arithmetic(Mathematical) operators

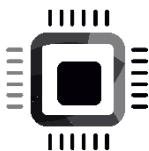
- + addition**
- subtraction**
- * multiplication**
- / division**
- % modulus**

Modulus produces the remainder from division.

For example:

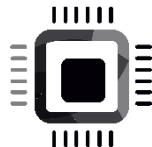
14 % 4 evaluates to 2.

14 divided by 4 yields a remainder of 2.



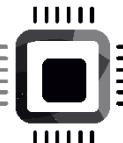
```
uint32_t value ;  
value = 2 + 3 * 4;
```

value = 20 or value = 14 ?



Operator Precedence

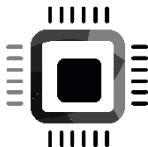
Operator precedence rules determine which mathematical operation takes place first, i.e. takes precedence over others. Parentheses, (), may be used to force an expression to a higher precedence



Precedence	Operator	Description	Associativity
1	<code>++ --</code> <code>()</code> <code>[]</code> <code>.</code> <code>-></code> <code>(type) { list }</code>	Suffix/postfix increment and decrement Function call Array subscripting Structure and union member access Structure and union member access through pointer Compound literal(C99)	Left-to-right
2	<code>++ --</code> <code>+ -</code> <code>! ~</code> <code>(type)</code> <code>*</code> <code>&</code> <code>sizeof</code> <code>_Alignof</code>	Prefix increment and decrement ^[note 1] Unary plus and minus Logical NOT and bitwise NOT Type cast Indirection (dereference) Address-of Size-of ^[note 2] Alignment requirement(C11)	Right-to-left
3	<code>* / %</code>	Multiplication, division, and remainder	Left-to-right
4	<code>+ -</code>	Addition and subtraction	Left-to-right
5	<code><< >></code>	Bitwise left shift and right shift	Left-to-right
6	<code>< <=</code> <code>> >=</code>	For relational operators <code><</code> and <code>≤</code> respectively For relational operators <code>></code> and <code>≥</code> respectively	Left-to-right
7	<code>== !=</code>	For relational <code>=</code> and <code>≠</code> respectively	Left-to-right
8	<code>&</code>	Bitwise AND	Left-to-right
9	<code>^</code>	Bitwise XOR (exclusive or)	Left-to-right
10	<code> </code>	Bitwise OR (inclusive or)	Left-to-right
11	<code>&&</code>	Logical AND	Left-to-right
12	<code> </code>	Logical OR	Left-to-right
13	<code>? :</code>	Ternary conditional ^[note 3]	Right-to-Left
14 ^[note 4]	<code>=</code> <code>+= -=</code> <code>*= /= %=</code> <code><<= >>=</code> <code>&= ^= =</code>	Simple assignment Assignment by sum and difference Assignment by product, quotient, and remainder Assignment by bitwise left shift and right shift Assignment by bitwise AND, XOR, and OR	Right-to-Left
15	<code>,</code>	Comma	Left-to-right

Don't try to memorize this table. Use `()` if you want to promote the precedence of an operator of your choice in an expression.

Image source
<https://en.cppreference.com>



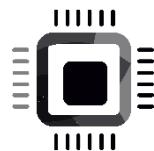
Examples

$2 + 3 * 4$ evaluates to 14 (multiplication first, addition second).

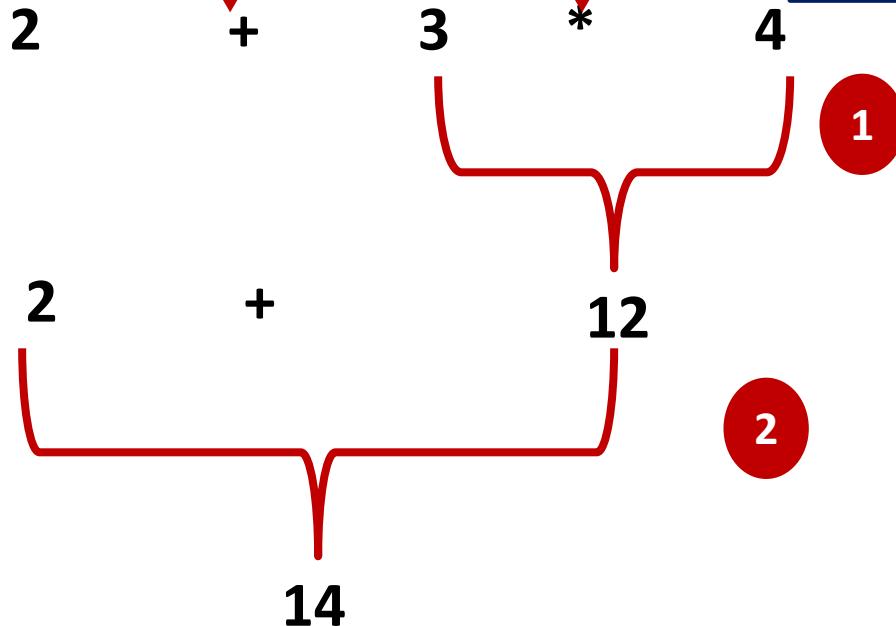
$(2 + 3) * 4$ evaluates to 20 (inside parentheses first, multiplication second).

$(2 * (4 + (6/2)))$ evaluates to 14 (inside parentheses first, work outward).

$4 * 5/2 * 5$ evaluates to 50 (start left, move right).

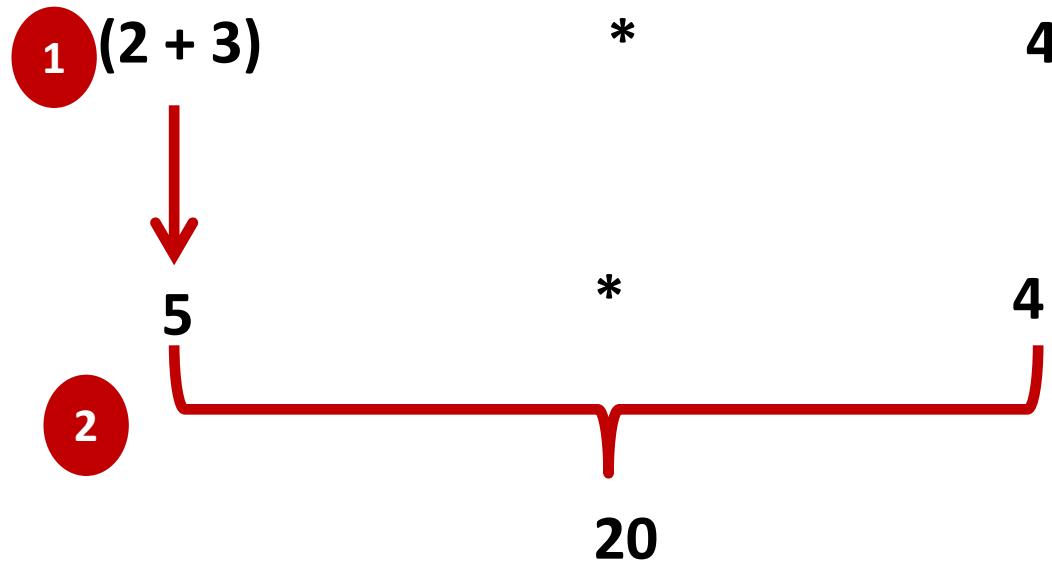


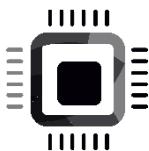
* : higher precedence
+ : lower precedence



$(2 + 3) * 4$

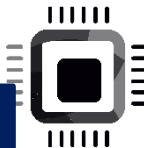
() : higher precedence
* : lower precedence



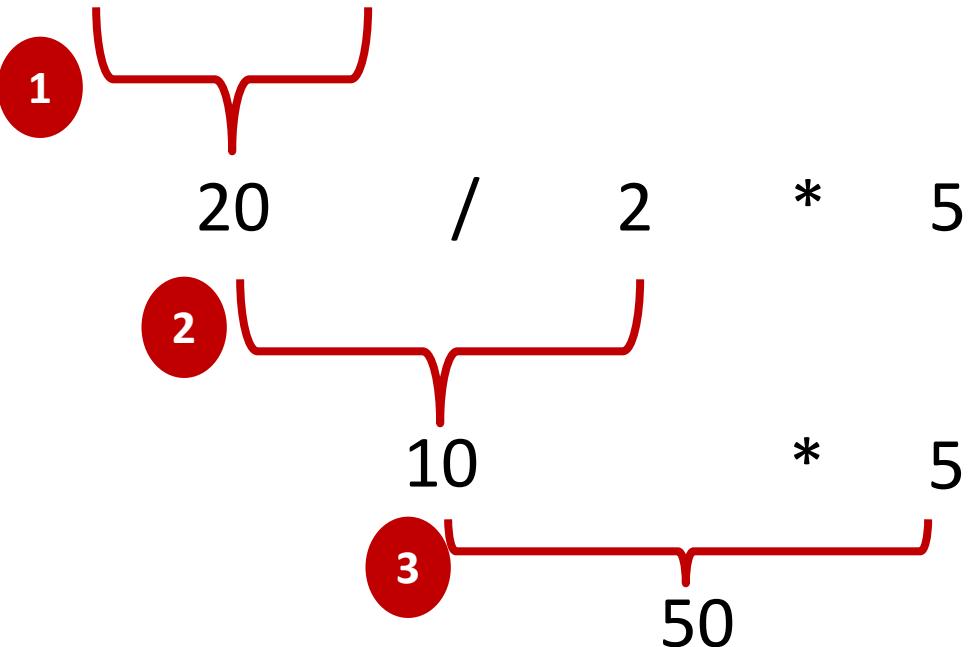


4 * 5/2 * 5

2 or 50 ??

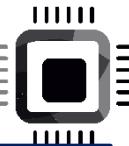


$$4 * 5 / 2 * 5$$



* and / have the same precedence.
Associativity: Left to Right

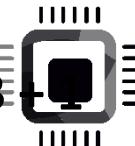
Associativity is used to evaluate the expression when there are two or more operators of the same precedence is present in an expression.



Precedence	Operator	Description	Associativity
1	<code>++ --</code> <code>()</code> <code>[]</code> <code>.</code> <code>-></code> <code>(type) { list }</code>	Suffix/postfix increment and decrement Function call Array subscripting Structure and union member access Structure and union member access through pointer Compound literal(C99)	Left-to-right
2	<code>++ --</code> <code>+ -</code> <code>! ~</code> <code>(type)</code> <code>*</code> <code>&</code> <code>sizeof</code> <code>_Alignof</code>	Prefix increment and decrement ^[note 1] Unary plus and minus Logical NOT and bitwise NOT Type cast Indirection (dereference) Address-of Size-of ^[note 2] Alignment requirement(C11)	Right-to-left
3	<code>* / %</code>	Multiplication, division, and remainder	Left-to-right
4	<code>+ -</code>	Addition and subtraction	Left-to-right
5	<code><< >></code>	Bitwise left shift and right shift	Left-to-right
6	<code>< <=</code> <code>> >=</code>	For relational operators <code><</code> and <code>≤</code> respectively For relational operators <code>></code> and <code>≥</code> respectively	Left-to-right
7	<code>== !=</code>	For relational <code>=</code> and <code>≠</code> respectively	Left-to-right
8	<code>&</code>	Bitwise AND	Left-to-right
9	<code>^</code>	Bitwise XOR (exclusive or)	Left-to-right
10	<code> </code>	Bitwise OR (inclusive or)	Left-to-right
11	<code>&&</code>	Logical AND	Left-to-right
12	<code> </code>	Logical OR	Left-to-right
13	<code>? :</code>	Ternary conditional ^[note 3]	Right-to-Left
14 ^[note 4]	<code>=</code> <code>+= -=</code> <code>*= /= %=</code> <code><<= >>=</code> <code>&= ^= =</code>	Simple assignment Assignment by sum and difference Assignment by product, quotient, and remainder Assignment by bitwise left shift and right shift Assignment by bitwise AND, XOR, and OR	Right-to-Left
15	<code>,</code>	Comma	Left-to-right

Don't try to memorize this table. Use `()` if you want to promote the precedence of an operator of your choice in an expression.

Image courtesy
<https://en.cppreference.com>



$$12 + 3 - 4 / 2 < 3 + 1$$

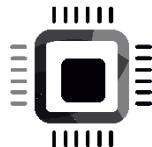
$$12 + 3 - 2 < 3 + 1$$

$$15 - 2 < 3 + 1$$

$$13 < 3 + 1$$

$$13 < 4$$

$$13 < 4$$



All arithmetical operators are binary operators. i.e., they need at least two operands to operate.

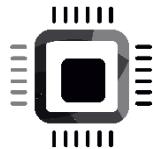
+ addition

- subtraction

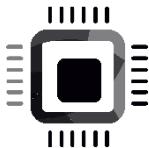
*** multiplication**

/ division

% modulus



Unary operator in ‘C’



Operand

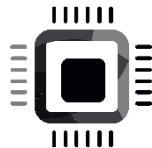
X++

Unary increment operator

X--

Unary decrement operator

Unary operators : ++, --

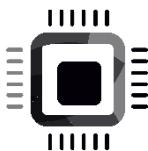


++ is a unary operator and it can appear on either side of an expression

```
uint32_t x;  
x++; //x is incremented by 1  
++x; //x is incremented by 1
```

The **++ (increment)** operator adds 1 to the value of the operand and updates the operand

Note : when the operand is pointer variable the behavior will be different and we will discuss later



++ is a unary operator and it can appear on either side of an expression

uint32_t x,y;

x = 5;

y = ++x;

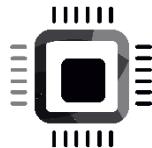
y=6 , x = 6

uint32_t n,m;

n = 5;

m = n++;

m=5, x = 6



++ is a unary operator and it can appear on either side of an expression

uint32_t x,y;

x = 5;

y = **++x;**



Pre-incrementing

y=6 , x = 6

*First, value of x will be incremented by 1
Then value of x will be assigned to y*

uint32_t n,m;

n = 5;

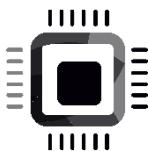
m = **n++;**



Post-incrementing

m=5, x = 6

*First value of x will be assigned to m
Then value of x will be incremented by 1*



-- is a unary operator and it can appear on either side of an expression

`uint32_t x,y;`

`x = 5;`

`y = --x;`



Pre-decrementing

`y=4 , x = 4`

`uint32_t n,m;`

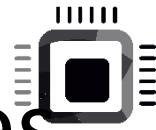
`n = 5;`

`m = n--;`



Post-decrementing

`m=5, n = 4`

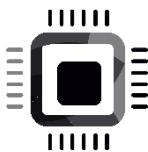


Unary operators with pointer variables

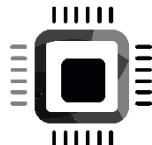
```
uint32_t *pAddress = (uint32_t*) 0xFFFF0000;
```

```
pAddress = pAddress +1 ; //this is arithmetic add operation  
(result : pAddress = 0xFFFF0004)
```

```
pAddress++; //This is unary increment operation  
(pAddress = 0xFFFF0004)
```



Relational Operators in ‘C’

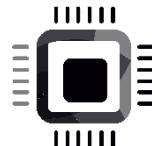


Relational Operators in ‘C’

- ✓ Relational operators do some kind of evaluation on the operands and then return value 1 (for true) or 0 (for false).

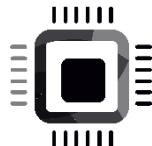
- ✓ Relational operators are binary operators because they require two operands to operate.

- ✓ The relational operators are evaluated left to right.



Relational Operators available in 'C'

==	Equal to	$a==b$ returns 1 if a and b are the same
>	Greater than	$a>b$ returns 1 if a is larger than b
<	Less than	$a<b$ returns 1 if a is smaller than b
\geq	Greater than or equal to	$a\geq b$ returns 1 if a is larger than or equal to b
\leq	Less than or equal to	$a\leq b$ returns 1 if a is smaller than or equal to b
!=	Not equal to	$a!=b$ returns 1 if a and b not the same



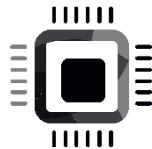
True and False in ‘C’

- ✓ In ‘C’ Zero is interpreted as false and anything non-zero is interpreted as true.

- ✓ Expressions using relational operators evaluate to a value of either **TRUE (1)** or **FALSE (0)**.

- ✓ Relational expressions are often used within **if** and **while** statements.

Examples

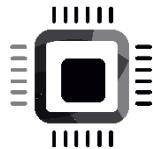


A = 10, B=20

C= (A == B); If A and B are the same value, then this expression evaluates to be TRUE(1) otherwise FALSE(0). So, here 0 will be stored in C;

C= (A <= B); If A is smaller than or equal to B, then this expression evaluates to be TRUE(1) otherwise FALSE(0). So, here 1 will be stored in C;

Examples

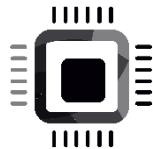


A = 10, B=20

C= (A != B); What happens here ?

C= (A < B); What happens here ?

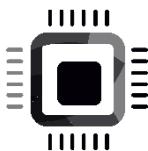
Examples



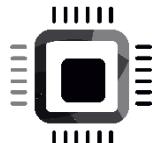
A = 10, B=20

C= (A != B); What happens here ? C= 1

C= (A < B); What happens here ? C= 1



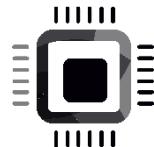
Logical Operators in ‘C’



Logical Operators in 'C'

- The logical operators perform logical-AND (**&&**) ,logical-OR (**||**) and logical –NOT (**!**) operations.
- These are binary operators except NOT (!).

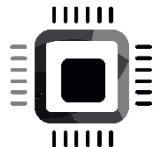
&& !	And Or Not (unary)	a&&b returns 1 if a is nonzero and b is nonzero a b returns 1 if a is nonzero or b is nonzero !a returns 1 if a is zero
--	--------------------------	--



logical AND Operator : **&&**

- ✓ The **logical-AND** operator produces the value 1 if both operands have nonzero values .
- ✓ If either operand is equal to 0, the result is 0. If the first operand of a logical-AND operation is equal to 0, the second operand is not evaluated.

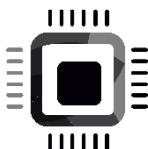
logical AND Operator : **&&**



A = -10, B=20;

C= (A && B); // true && true

C = 1 //true



Truth table

AND

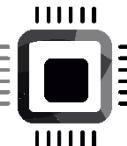
a	b	a && b
T	T	T
T	F	F
F	T	F
F	F	F

OR

a	b	a b
T	T	T
T	F	T
F	T	T
F	F	F

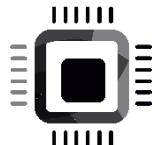
NOT

a	!a
T	F
F	T



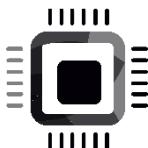
Precedence	Operator	Description	Associativity
1	<code>++ --</code> <code>()</code> <code>[]</code> <code>.</code> <code>-></code> <code>(type) { list }</code>	Suffix/postfix increment and decrement Function call Array subscripting Structure and union member access Structure and union member access through pointer Compound literal(c99)	Left-to-right
2	<code>++ --</code> <code>+ -</code> <code>! ~</code> <code>(type)</code> <code>*</code> <code>&</code> <code>sizeof</code> <code>_Alignof</code>	Prefix increment and decrement[note 1] Unary plus and minus Logical NOT and bitwise NOT Type cast Indirection (dereference) Address-of Size-of[note 2] Alignment requirement(c11)	Right-to-left
3	<code>* / %</code>	Multiplication, division, and remainder	Left-to-right
4	<code>+ -</code>	Addition and subtraction	
5	<code><< >></code>	Bitwise left shift and right shift	
6	<code>< <=</code> <code>> >=</code>	For relational operators <code><</code> and <code>≤</code> respectively For relational operators <code>></code> and <code>≥</code> respectively	
7	<code>== !=</code>	For relational <code>=</code> and <code>≠</code> respectively	
8	<code>&</code>	Bitwise AND	
9	<code>^</code>	Bitwise XOR (exclusive or)	
10	<code> </code>	Bitwise OR (inclusive or)	
11	<code>&&</code>	Logical AND	
12	<code> </code>	Logical OR	
13	<code>? :</code>	Ternary conditional[note 3]	Right-to-Left
14[note 4]	<code>=</code>	Simple assignment	
	<code>+= -=</code>	Assignment by sum and difference	
	<code>*= /= %=</code>	Assignment by product, quotient, and remainder	
	<code><<= >>=</code>	Assignment by bitwise left shift and right shift	
	<code>&= ^= =</code>	Assignment by bitwise AND, XOR, and OR	
15	<code>,</code>	Comma	Left-to-right

Image courtesy
<https://en.cppreference.com>



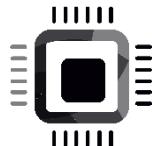
logical OR Operator : ||

- ✓ The logical-OR operator performs an inclusive-OR operation on its operands.
- ✓ The result is 0 if both operands have 0 values.
- ✓ If either operand has a nonzero value, the result is 1.
- ✓ If the first operand of a logical-OR operation has a nonzero value, the second operand is not evaluated.



logical NOT Operator : !

- ✓ It reverses the state of the operand.
- ✓ If a condition or expression is true, then Logical NOT operator will make it false



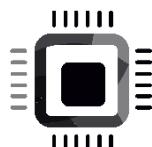
logical NOT Operator : !

```
x = 10; // state of the x is true
```

```
/*state of the x is reversed and stored back in x */
```

```
x = !x; // NOT of x ( NOT of true = false)
```

Result : x = 0;

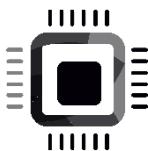


logical NOT Operator : !

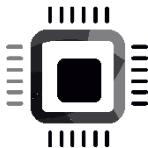
What is the value of A in the below expression ?

x = 7 , y = 9

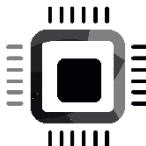
A = ! ((x>5) && (y<5));



Bitwise Operators in ‘C’



Operator	Type
<code>++, --</code>	Increment and decrement operator (Unary operator)
<code>+, -, *, /, %</code>	Arithmetic operators
<code><, <=, >, >=, ==, !=</code>	Relational operators
<code>&&, , !</code>	Logical Operators
<code>&, , <<, >>, ~, ^</code>	Bitwise operators
<code>=, +=, -=, *=, /=, %=</code>	Assignment operators
<code>?:</code>	Conditional operators



Bitwise Operators in 'C'



(bitwise AND)



(bitwise Right Shift)



(bitwise OR)



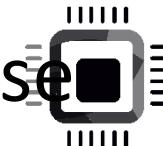
(bitwise NOT) (Negation)



(bitwise Left Shift)



(bitwise XOR)



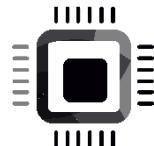
Difference between Logical operator and bitwise operator

- **&&** 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;
(Logical)

C = A & B;
(Bitwise)

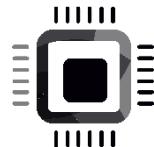
C=1;

A
&
B

b00101000
b00011110

C= b00001000

Bitwise operation

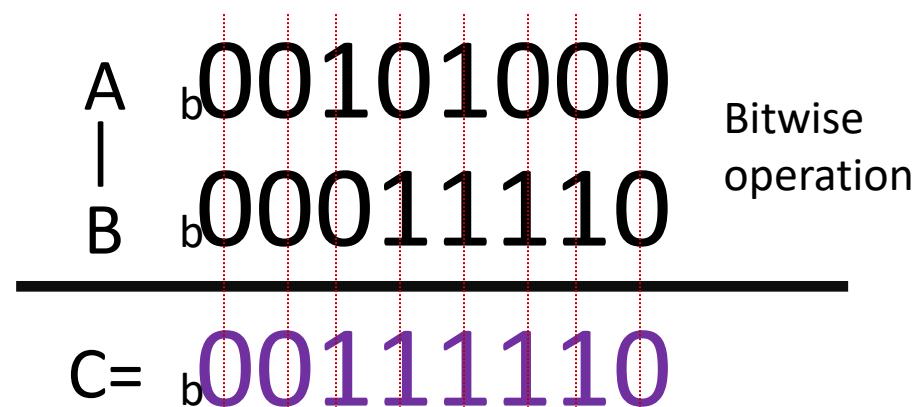


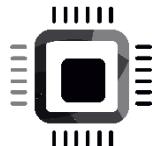
char A = 40; char B = 30; char C;

C = A || B;

C=1;

C = A | B;





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

C = ! A;

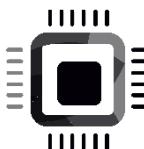
C=0;

C = \sim A;

A b 00101000
C = \sim A b 11010111

C = -41

Bitwise
operation



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

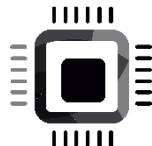
C = A ^ B;

Truth table for XOR Gate

INPUTS		OUTPUTS
A	B	$Y = A \wedge B$
0	0	0
0	1	1
1	0	1
1	1	0

Bitwise
operation

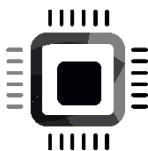
A	\wedge	B	=	C
00101000		00011110		00110110



Applicability of bitwise operations

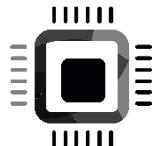
In an Embedded C program, most of the time you will be doing,

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



Exercise

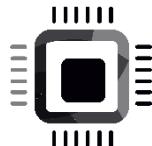
- Write a program which takes 2 integers from the user , computes bitwise &,|,^ and ~ and prints the result.



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

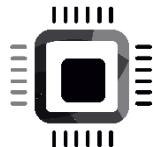
- Write a program to find out whether a user entered integer is even or odd.
- Print an appropriate message on the console.
- Use testing of bits logic

46 (0x2E)
lsb
b00101110

Number is EVEN

47 (0x2F)
lsb
b00101111

Number is ODD

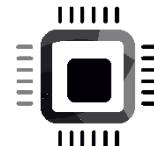


Test the least significant bit of the number using bitwise operation

If the lsb is zero, then number is EVEN

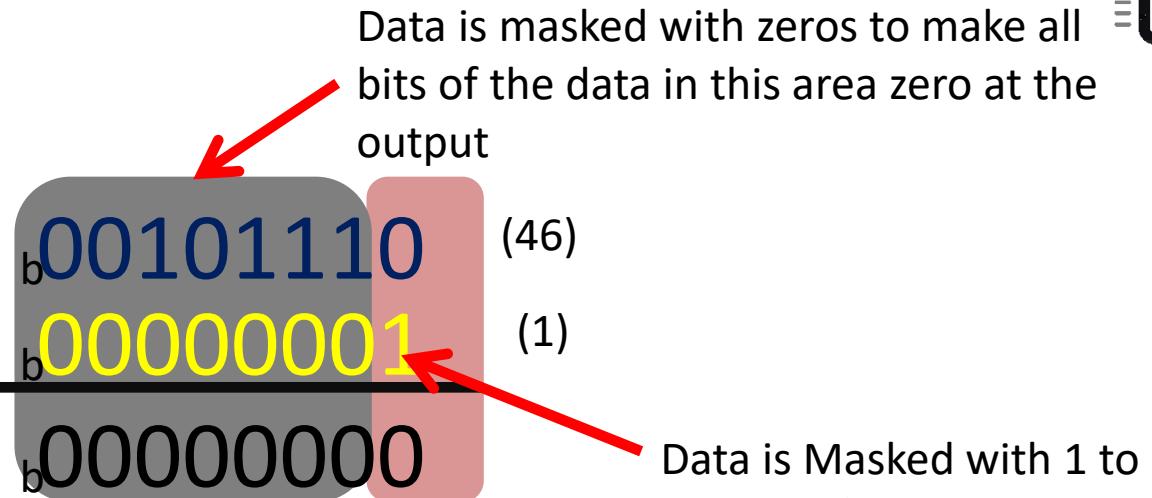
If the lsb is one, then number is ODD

Bit-Masking



[input] number
 &
[mask] Mask_value

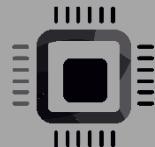
[output]



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

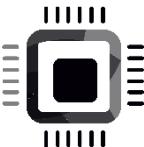


[data]	00101110
&	
[Mask]	00000010
<hr/>	
[output]	00000010

[data]	10101110
&	
[Mask]	10000000
<hr/>	
[output]	10000000

[data]	00101110
&	
[Mask]	00000011
<hr/>	
[output]	00000010

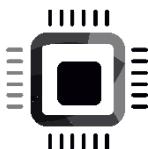
[data]	00101110
&	
[Mask]	00010000
<hr/>	
[output]	00000000



EVEN or ODD using Bitwise operation

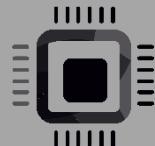
```
If ( number & 1 )  
{  
    print(number odd)  
}else  
{  
    print(Number even)  
}
```

↗ Mask value



Exercise: Setting of bits

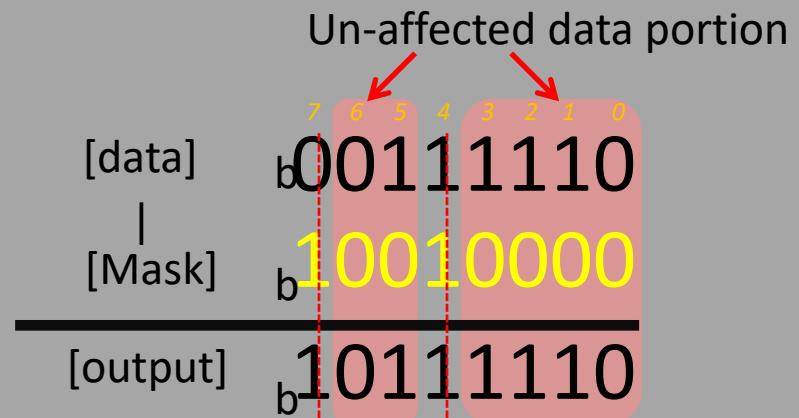
- Write a program to set(make bit state to 1) 4th and 7th-bit position of a given number and print the result.



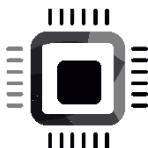
Exercise: Setting of bits

	7	6	5	4	3	2	1	0
[data]	b	0	0	1	1	1	1	0
& [Mask]	b	1	0	0	1	0	0	0
[output]	b	0	0	0	1	0	0	0

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

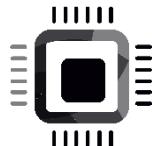


'|' is used to 'SET' not to 'TEST'



Exercise: Clearing of bits

- Write a program to clear(make bit state to 0) 4th, 5th, 6th, bit positions of a given number and print the result.



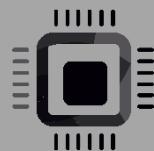
Exercise: Clearing of bits

Which bit wise operation do you use to clear the given bit position of the data ?

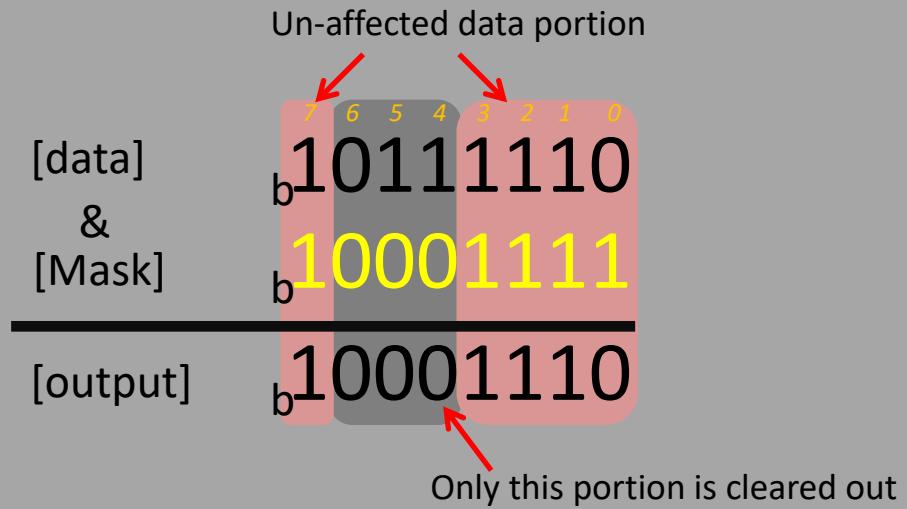
& or | ?

'&' is used to 'TEST and CLEAR' not to 'SET'

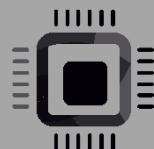
'|' is used to 'SET' not to 'TEST'



Exercise: Clearing of bits



'&' is used to 'TEST and CLEAR' not to 'SET'



Exercise: Clearing of bits

Method-I

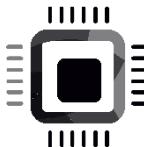
	Un-affected data portion							
[data] & [Mask]	b	1	0	1	1	1	1	0
	b	1	0	0	0	1	1	1
[output]	Only this portion is cleared out							
b	1	0	0	0	1	1	0	0

Method-II

	Un-affected data portion							
[data] & [Mask]	b	1	0	1	1	1	1	0
	b	~(0	1	1	0	0	0)
[output]	Only this portion is cleared out							
b	1	0	0	0	1	1	0	0

*Negate (Bitwise NOT) the mask value first
and then perform bitwise AND (&)*

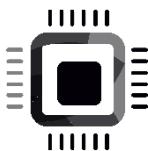
Toggling of bits



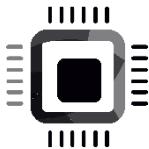
[Led_state]	00000001
^	
[mask]	00000001
<hr/>	
[Led_state]	00000000
^	
[mask]	00000001
<hr/>	
[Led_state]	00000001
^	
[mask]	00000001
<hr/>	
[Led_state]	00000000
^	
[mask]	00000001
<hr/>	
[Led_state]	00000001

$\text{Led_state} = \text{Led_state} \wedge 0x01$

Output toggles

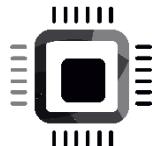


Applicability of bitwise operations in Embedded Systems



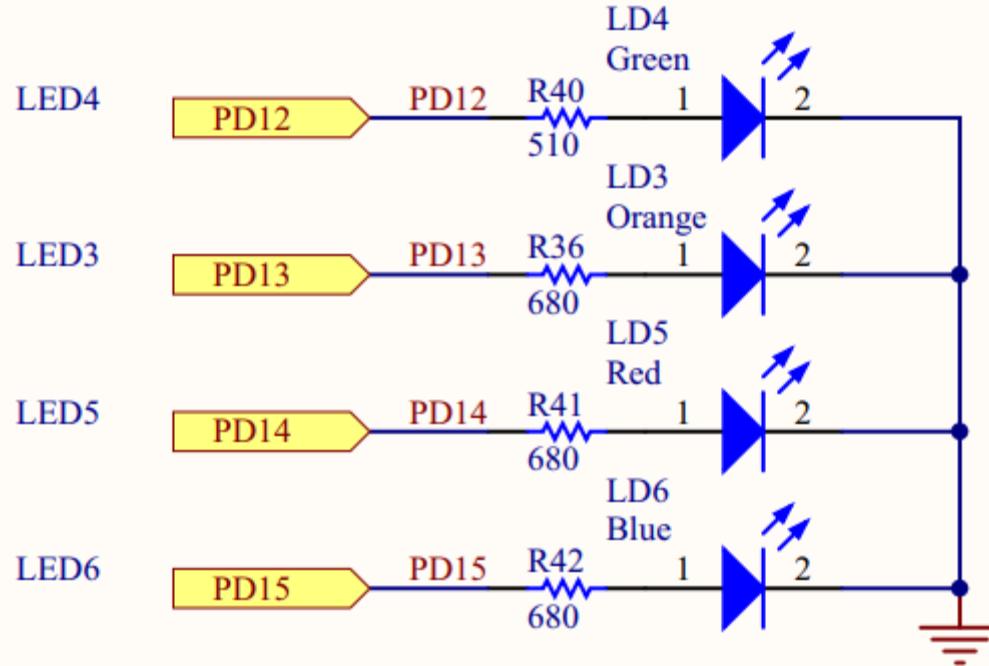
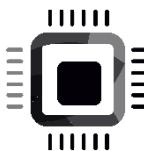
Exercise

- 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



LEDs



PORT-A

PA0	PA0	23
PA1	PA1	24
PA2	PA2	25
PA3	PA3	26
PA4	PA4	29
PA5	PA5	30
PA6	PA6	31
PA7	PA7	32
PA8	PA8	67
PA9	PA9	68
PA10	PA10	69
PA11	PA11	70
PA12	PA12	71
PA13	PA13	72
PA14	PA14	76
PA15	PA15	77

PORT-B

PB0	PB0	35
PB1	PB1	36
PB2	PB2	37
PB3	PB3	89
PB4	PB4	90
PB5	PB5	91
PB6	PB6	92
PB7	PB7	93
PB8	PB8	95
PB9	PB9	96
PB10	PB10	47
PB11	PB11	48
PB12	PB12	51
PB13	PB13	52
PB14	PB14	53
PB15	PB15	54

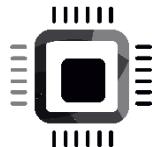
PORT-C

PC0	PC0	15
PC1	PC1	16
PC2	PC2	17
PC3	PC3	18
PC4	PC4	33
PC5	PC5	34
PC6	PC6	63
PC7	PC7	64
PC8	PC8	65
PC9	PC9	66
PC10	PC10	78
PC11	PC11	79
PC12	PC12	80
PC13	PC13	7

PORT-E

PORTD

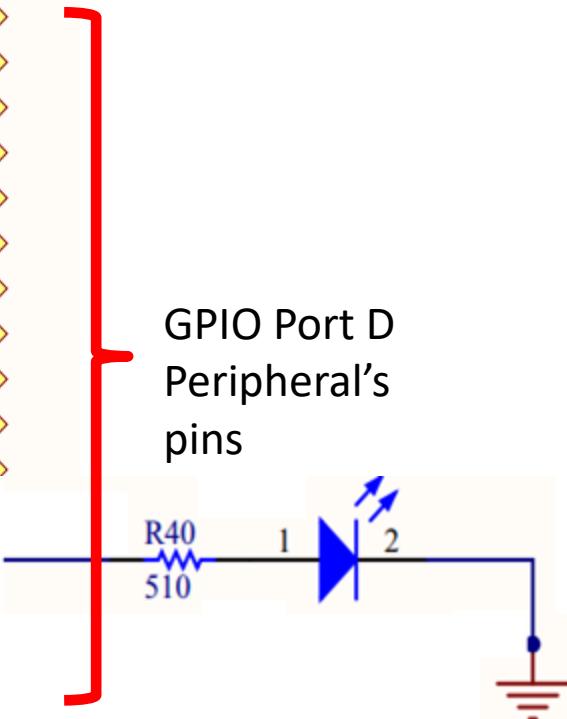
In STM32Fx based MCUs , each port has 16 pins where you can connect external peripherals . (LED, Display , button, Bluetooth transceiver , external memory (e.g. EEPROM), Joy stick , keypad ,etc)

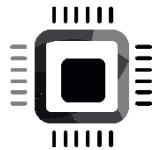


How do you access 12th pin of GPIO PORT-D peripheral from software ?

PB0	PB0	35	PB0	PD0	81	PD0
PB1	PB1	36	PB1	PD1	82	PD1
PB2	PB2	37	PB2	PD2	83	PD2
PB3	PB3	89	PB3	PD3	84	PD3
PB4	PB4	90	PB4	PD4	85	PD4
PB5	PB5	91	PB5	PD5	86	PD5
PB6	PB6	92	PB6	PD6	87	PD6
PB7	PB7	93	PB7	PD7	88	PD7
PB8	PB8	95	PB8	PD8	55	PD8
PB9	PB9	96	PB9	PD9	56	PD9
PB10	PB10	47	PB10	PD10	57	PD10
PB11	PB11	48	PB11	PD11	58	PD11
PB12	PB12	51	PB12	PD12	59	PD12
PB13	PB13	52	PB13	PD13	60	PD13
PB14	PB14	53	PB14	PD14	61	PD14
PB15	PB15	54	PB15	PD15	62	PD15

GPIO Port D
Peripheral's
pins

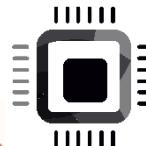




Now your goal is to control the I/O pin PD12's state either HIGH or LOW through software to make LED turn ON or OFF.

PD12 : 12th pin of the GPIO PORT D peripheral

**GPIO: General Purpose Input Output*

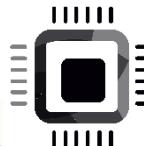


It also has set of registers which are used to control pin's mode, state and other functionalities



MICROCONTROLLER

81	PD0	PD0
82	PD1	PD1
83	PD2	PD2
84	PD3	PD3
85	PD4	PD4
86	PD5	PD5
87	PD6	PD6
88	PD7	PD7
55	PD8	PD8
56	PD9	PD9
57	PD10	PD10
58	PD11	PD11
59	PD12	PD12
60	PD13	PD13
61	PD14	PD14
62	PD15	PD15

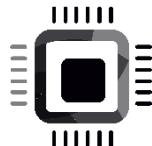


You can access the registers of this peripheral using memory addresses , hence you can also say that this peripheral registers are memory mapped.

GPIO D
peripheral
And its registers

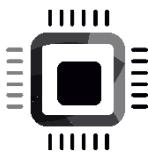
MICROCONTROLLER

81	PD0	PD0
82	PD1	PD1
83	PD2	PD2
84	PD3	PD3
85	PD4	PD4
86	PD5	PD5
87	PD6	PD6
88	PD7	PD7
55	PD8	PD8
56	PD9	PD9
57	PD10	PD10
58	PD11	PD11
59	PD12	PD12
60	PD13	PD13
61	PD14	PD14
62	PD15	PD15

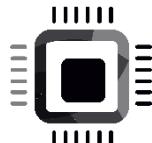


Memory mapped I/O

- IO pins are controlled using peripheral registers which are mapped on to processor addressable memory locations
- What are processor addressable memory locations ?

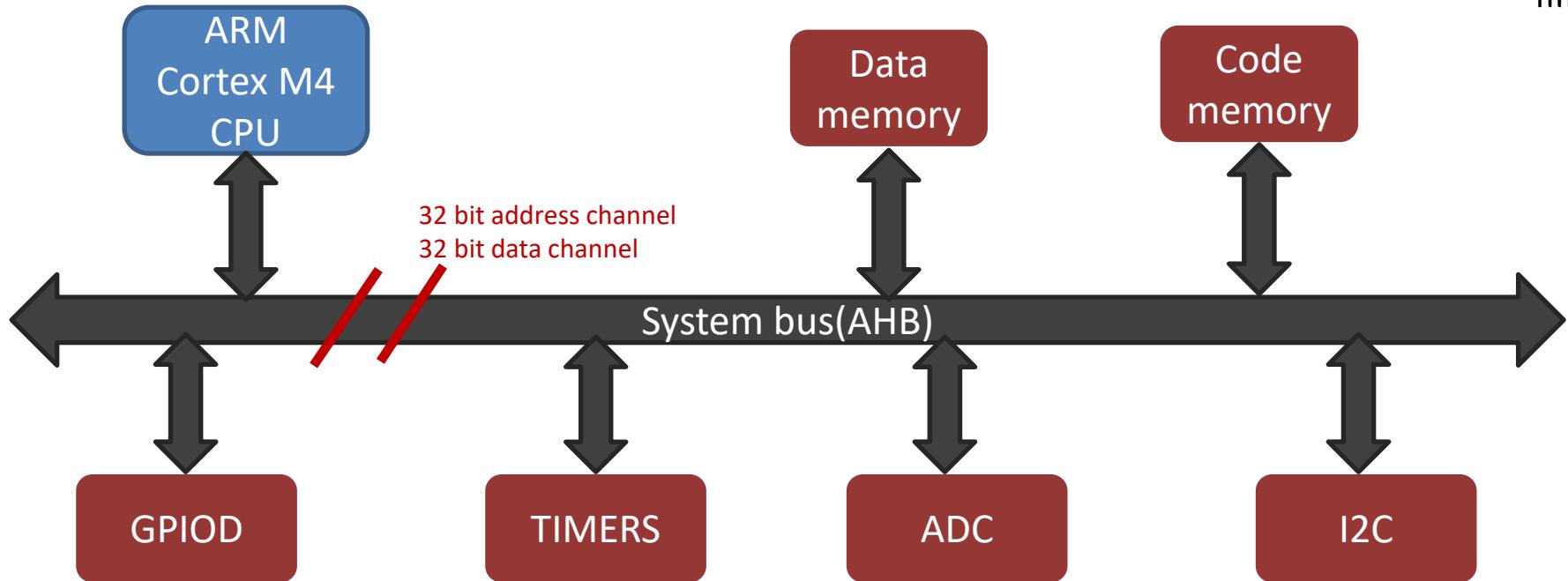
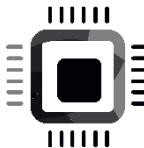


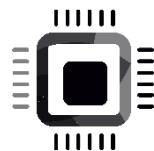
What are processor addressable memory locations ?



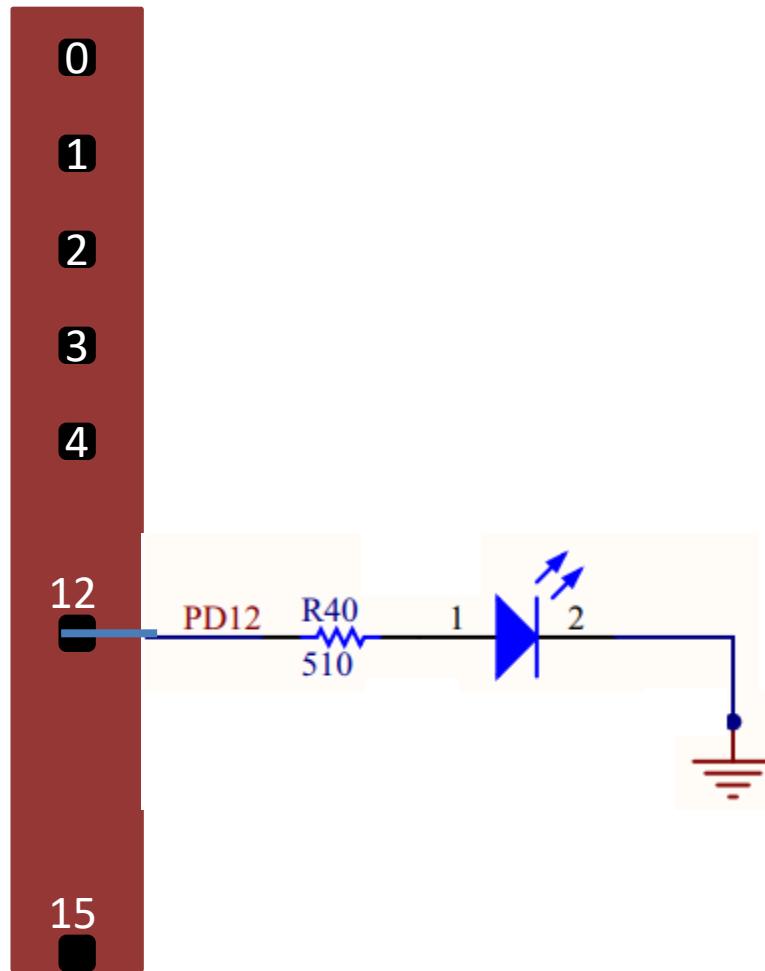
Memory Map of the processor

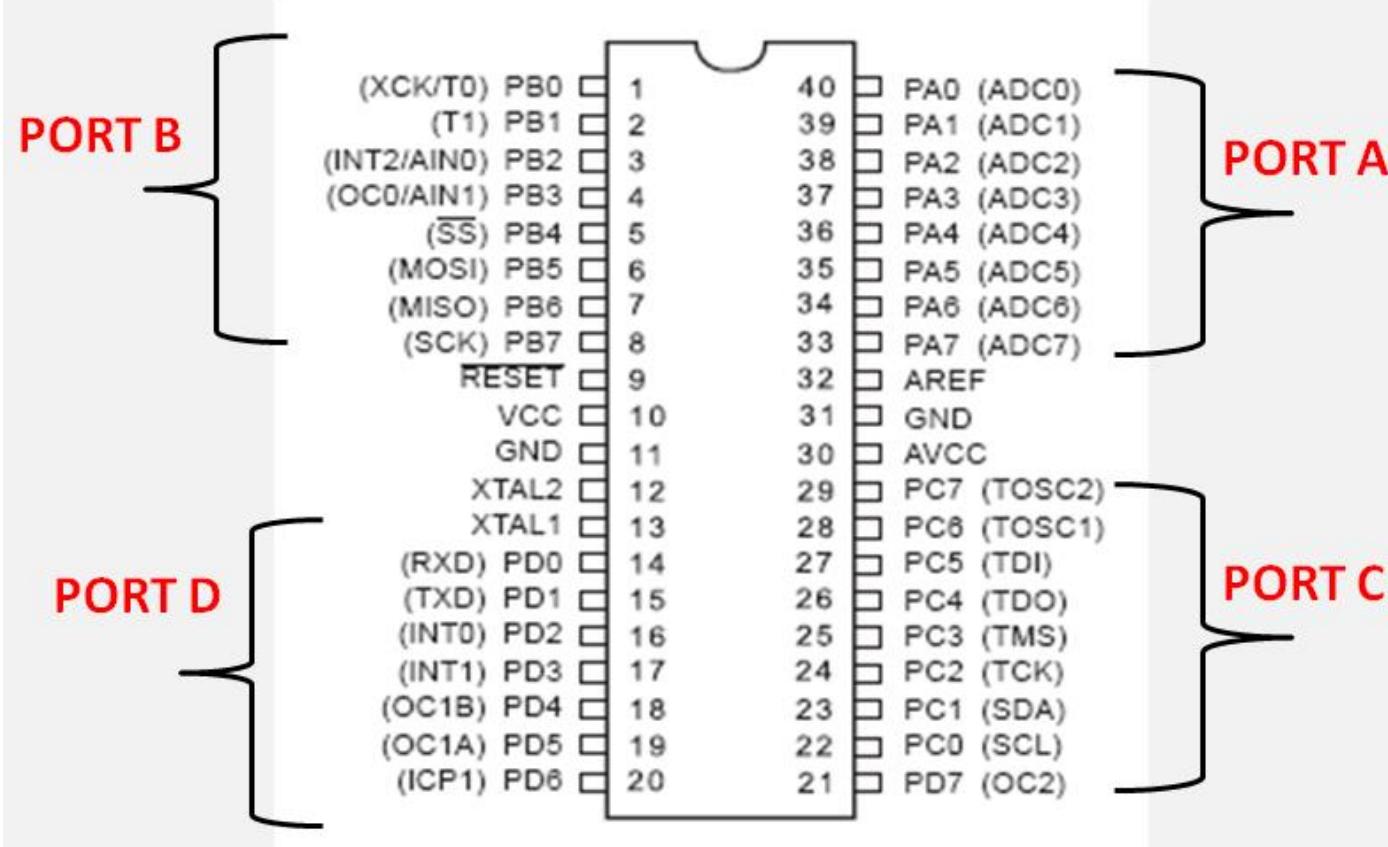
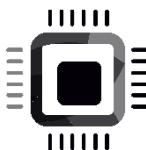
- Memory map explains mapping of different peripheral registers in the processor addressable memory location range
- Processor addressable memory location range depends upon size of the address bus.

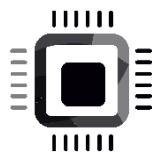




GPIO PORT-D Peripheral





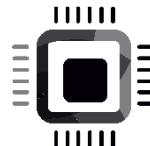


Processor addressable memory locations ?

Since address bus width is 32 bits,

Processor put address ranging from 0x0000_0000 to 0xFFFF_FFFF on the address bus.

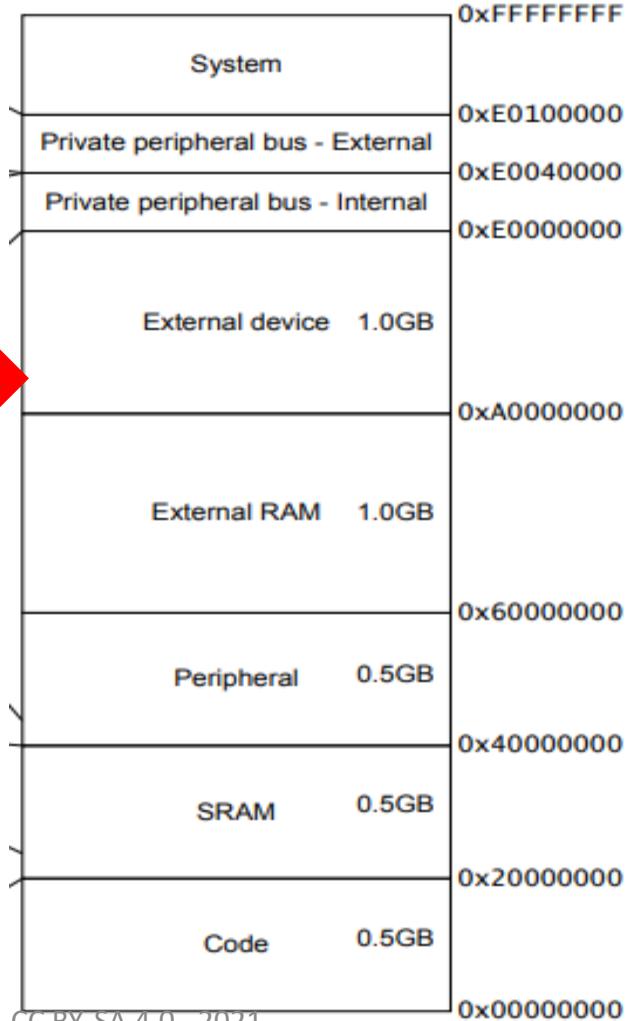
So, that means 4G(4,29,49,67,296)different addresses can be put on address bus

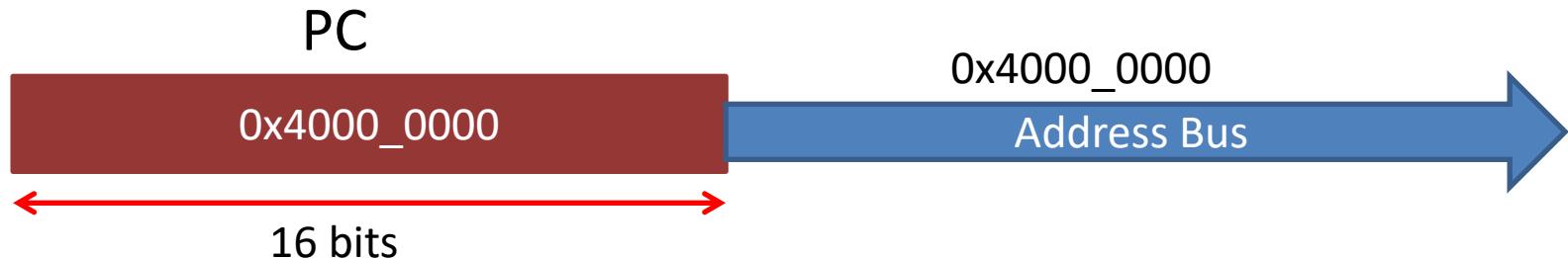
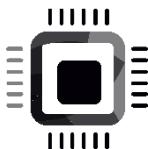


Memory map of ARM Cortex Mx processor

Program memory, data memory, registers of various peripherals are organized within the same linear 4 G byte of address space.

This is a generic memory map which must be followed by all MCUs which use ARM cortex Mx processor



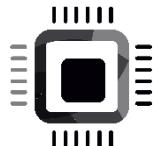


Here Processor wants to talk to a peripheral register which is mapped at the address 0x4000_0000.

Processor either wants to read from the register or write in to the register.

Which register of which peripheral ?

That depends on your microcontroller design.



Memory map of STM32 MCU

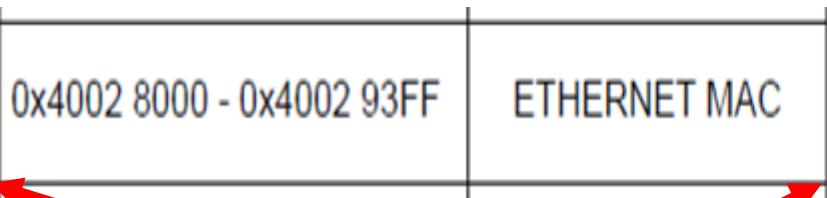
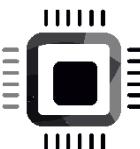
Memory map of the microcontroller you can refer from the datasheet or reference manual of the microcontroller



Table 1. STM32F4xx register boundary addresses (continued)

Boundary address	Peripheral	Bus	Register map
0x5006 0800 - 0x5006 0BFF	RNG	AHB2	Section 24.4.4: RNG register map on page 771
0x5006 0400 - 0x5006 07FF	HASH		Section 25.4.9: HASH register map on page 795
0x5006 0000 - 0x5006 03FF	CRYP		Section 23.6.13: CRYP register map on page 763
0x5005 0000 - 0x5005 03FF	DCMI		Section 15.8.12: DCMI register map on page 478
0x5000 0000 - 0x5003 FFFF	USB OTG FS		Section 34.16.6: OTG_FS register map on page 1326
0x4004 0000 - 0x4007 FFFF	USB OTG HS		Section 35.12.6: OTG_HS register map on page 1472
0x4002 B000 - 0x4002 BBFF	DMA2D		Section 11.5: DMA2D registers on page 352
0x4002 8000 - 0x4002 93FF	ETHERNET MAC		Section 33.8.5: Ethernet register maps on page 1236
0x4002 6400 - 0x4002 67FF	DMA2		Section 10.5.11: DMA register map on page 335
0x4002 6000 - 0x4002 63FF	DMA1		
0x4002 4000 - 0x4002 4FFF	BKPSRAM		
0x4002 3C00 - 0x4002 3FFF	Flash interface register		Section 3.9: Flash interface registers
0x4002 3800 - 0x4002 3BFF	RCC		Section 7.3.24: RCC register map on page 265

Activate Windows
Go to Settings to activate Windows.



Section 33.8.5: Ethernet register maps on page 1236

The registers used to control and configure the Ethernet MAC peripheral and registers used to read data from and write data into Ethernet MAC peripheral are mapped in the address range of 0x4002_8000 to 0x4002_93FF.



The ARM® Cortex®-M4 processor has a single 4 GB address space. The following table shows how this space is used on the LPC408x/407x.

Table 3. Memory usage and details

Address range	General Use	Address range details and description	
0x0000 0000 to 0x1FFF FFFF	On-chip non-volatile memory	0x0000 0000 - 0x0007 FFFF	For devices with 512 kB of flash memory.
		0x0000 0000 - 0x0003 FFFF	For devices with 256 kB of flash memory.
		0x0000 0000 - 0x0001 FFFF	For devices with 128 kB of flash memory.
		0x1000 0000 - 0x1000 FFFF	For devices with 64 kB of Main SRAM.
	On-chip SRAM	0x1000 0000 - 0x1000 7FFF	For devices with 32 kB of Main SRAM.
		0x1FFF 0000 - 0x1FFF 7FFF	8 kB Boot ROM with flash services.
	Boot ROM	0x1FFF 8000 - 0x1FFF 1FFF	16 kB Driver ROM
	On-chip SRAM (typically used for peripheral data)	0x2000 0000 - 0x2000 1FFF	Peripheral SRAM - bank 0 (first 8 kB)
		0x2000 2000 - 0x2000 3FFF	Peripheral SRAM - bank 0 (second 8 kB)
		0x2000 4000 - 0x2000 7FFF	Peripheral SRAM - bank 1 (16 kB)
		0x2008 0000 - 0x200B FFFF	See Section 2.3.1 for details
0x2000 0000 to 0x3FFF FFFF	SPIFI buffer space	0x2800 0000 - 0x28FF FFFF	SPIFI memory mapped access space
	APB Peripherals	0x4000 0000 - 0x4007 FFFF	APB0 Peripherals, up to 32 peripheral blocks of 16 kB each.
		0x4008 0000 - 0x400F FFFF	APB1 Peripherals, up to 32 peripheral blocks of 16 kB each.

Activate Windows

Get the latest updates from Microsoft to activate Windows.

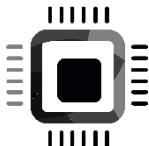


Redact



Type here to search

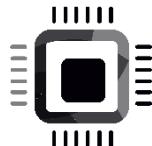




What is the address range of GPIOD peripheral registers?

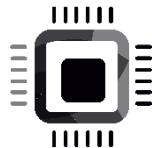
0x4002 0C00 - 0x4002 0FFF	GPIOD	AHB1
0x4002 0800 - 0x4002 0BFF	GPIOC	
0x4002 0400 - 0x4002 07FF	GPIOB	
0x4002 0000 - 0x4002 03FF	GPIOA	

*if you are using different MCU , then please refer to your device reference manual to obtain the correct address range



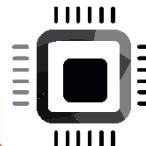
About peripheral registers

- All peripheral registers in STM32 microcontroller are of 32 bits wide
- Different peripherals have different number of peripheral registers.
- You should never assume about the address of the peripheral registers. Always you should refer to the device reference manual .



GPIOD Peripheral registers

- 1. GPIOD port mode register**
- 2. GPIOD port output type register**
- 3. GPIOD port output speed register**
- 4. GPIOD port pull-up/pull-down register**
- 5. GPIOD port input data register**
- 6. GPIOD port output data register**
- 7. GPIOD port bit set/reset register**
- 8. GPIOD port configuration lock register**
- 9. GPIOD alternate function low register**
- 10. GPIOD alternate function high register**

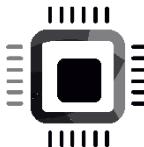


MICROCONTROLLER

GPIOD peripheral And its registers

GPIOD port mode register
GPIOD port output type register
GPIOD port output speed register
GPIOD port pull-up/pull-down register
GPIOD port input data register
GPIOD port output data register
GPIOD port bit set/reset register
GPIOD port configuration lock register
GPIOD alternate function low register
GPIOD alternate function high register

81	PD0	PD0
82	PD1	PD1
83	PD2	PD2
84	PD3	PD3
85	PD4	PD4
86	PD5	PD5
87	PD6	PD6
88	PD7	PD7
55	PD8	PD8
56	PD9	PD9
57	PD10	PD10
58	PD11	PD11
59	PD12	PD12
60	PD13	PD13
61	PD14	PD14
62	PD15	PD15



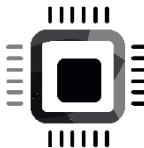
0x4002_0C03
0x4002_0C02
0x4002_0C01
0x4002_0C00

Most significant byte

Least significant byte

32 bit register

GPIOD port MODE register



0x4002_0C07
0x4002_0C06
0x4002_0C05
0x4002_0C04

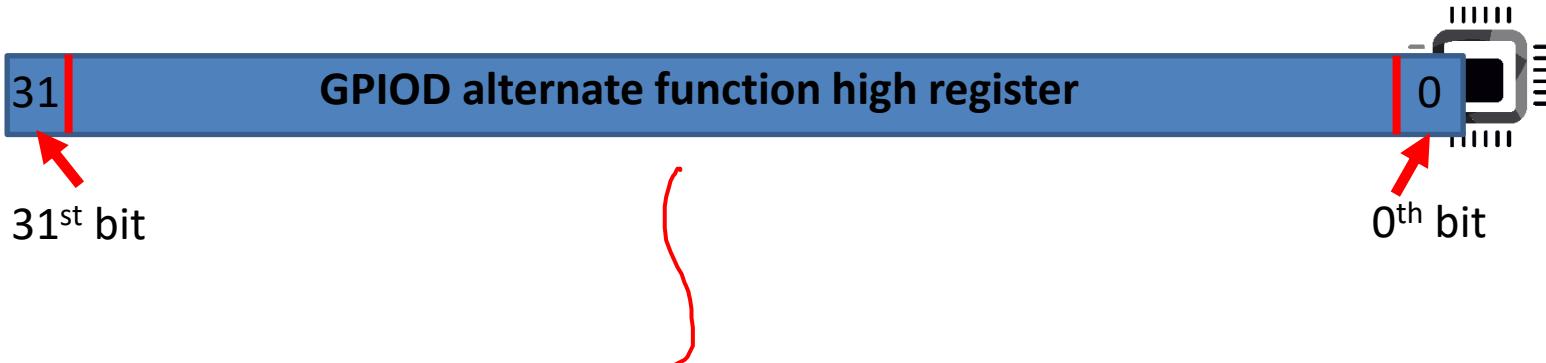
Most significant byte

Least significant byte

32 bit register

GPIOD port OUTPUT TYPE register

0x4002_0C24



0x4002_0C0C



0x4002_0C08



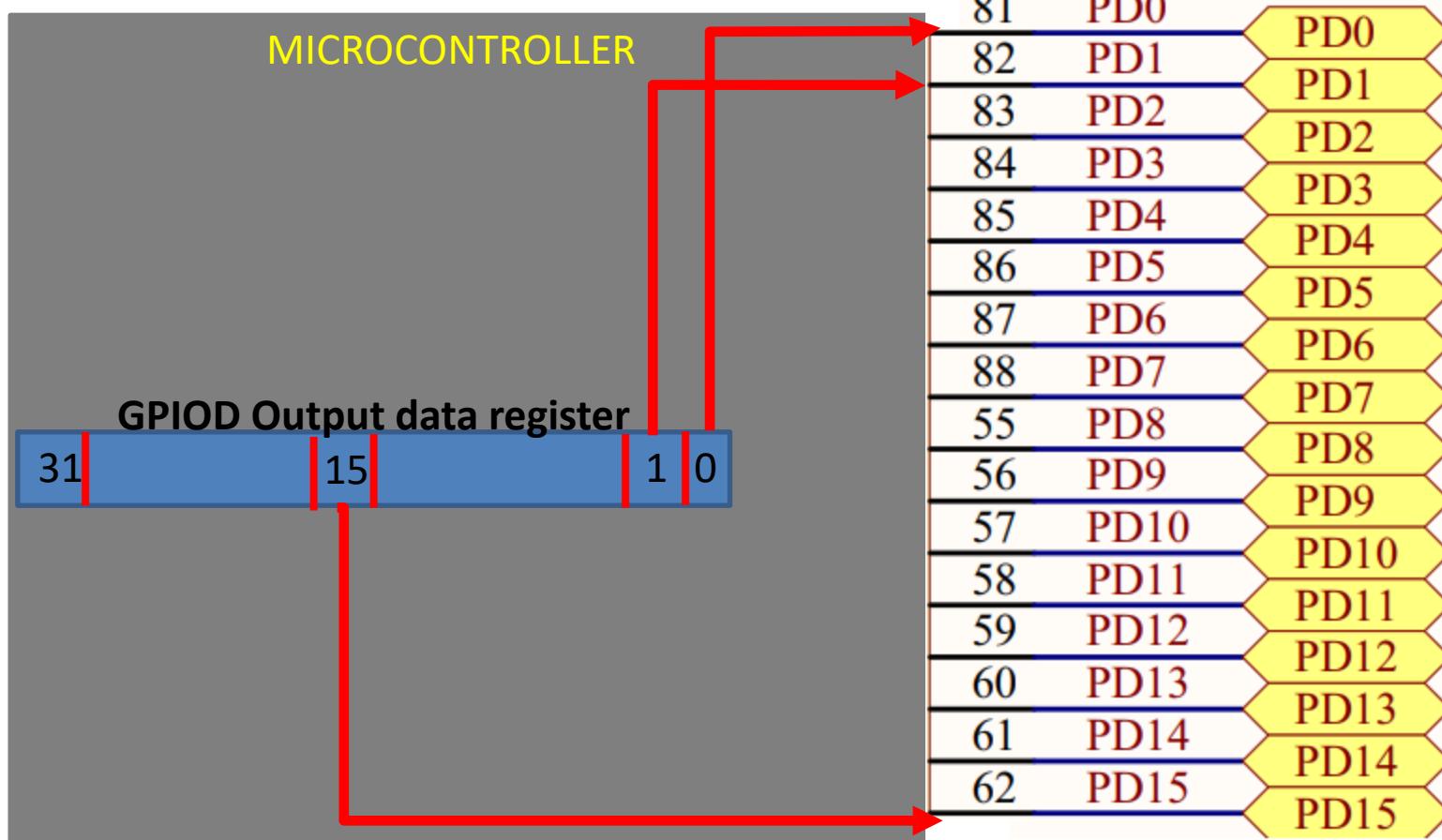
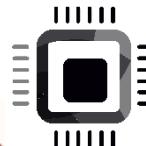
0x4002_0C04

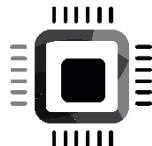


0x4002_0C00



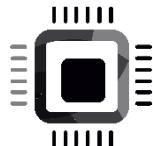
32 bits width





Procedure to turn on the LED

To write a code to turn on the LED is not that easy. That's what makes the embedded system programming so unique that you must know memory map, peripheral registers, peripheral register addresses, hardware connections, etc.



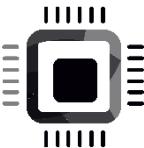
Procedure to turn on the LED

1) Identify the GPIO port(a peripheral) used to connect the LED

GPIOD

2) Identify the GPIO pin where the LED is connected

12

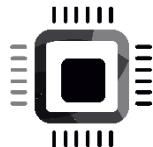


Procedure to turn on the LED

3) Activate the GPIOD peripheral (Enable the clock)

- Until you enable the clock for a peripheral, the peripheral is dead and it neither functions nor it takes any configuration values set by you.
- Once you active the clock for a peripheral, the peripheral is ready to take your configuration and control-related commands or arguments (configuration values)
- Note: For some microcontrollers, the peripheral may be ON by default, and you need not do any activation. (you should explore by the device datasheet for reference manual)

Procedure to turn on the LED



4) Configure the GPIO pin mode as output

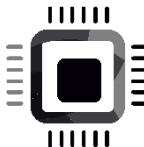
Since you are driving an LED (to ON or OFF), the operation mode of the **GPIO** pin has to be configured as **OUTPUT**.

5) Write to the GPIO pin

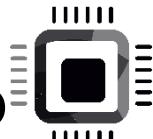
1 (HIGH) to make the GPIO pin state HIGH(3.3V)

0 (LOW) to make the GPIO pin state LOW(0V)

Procedure to turn on the LED : Summary



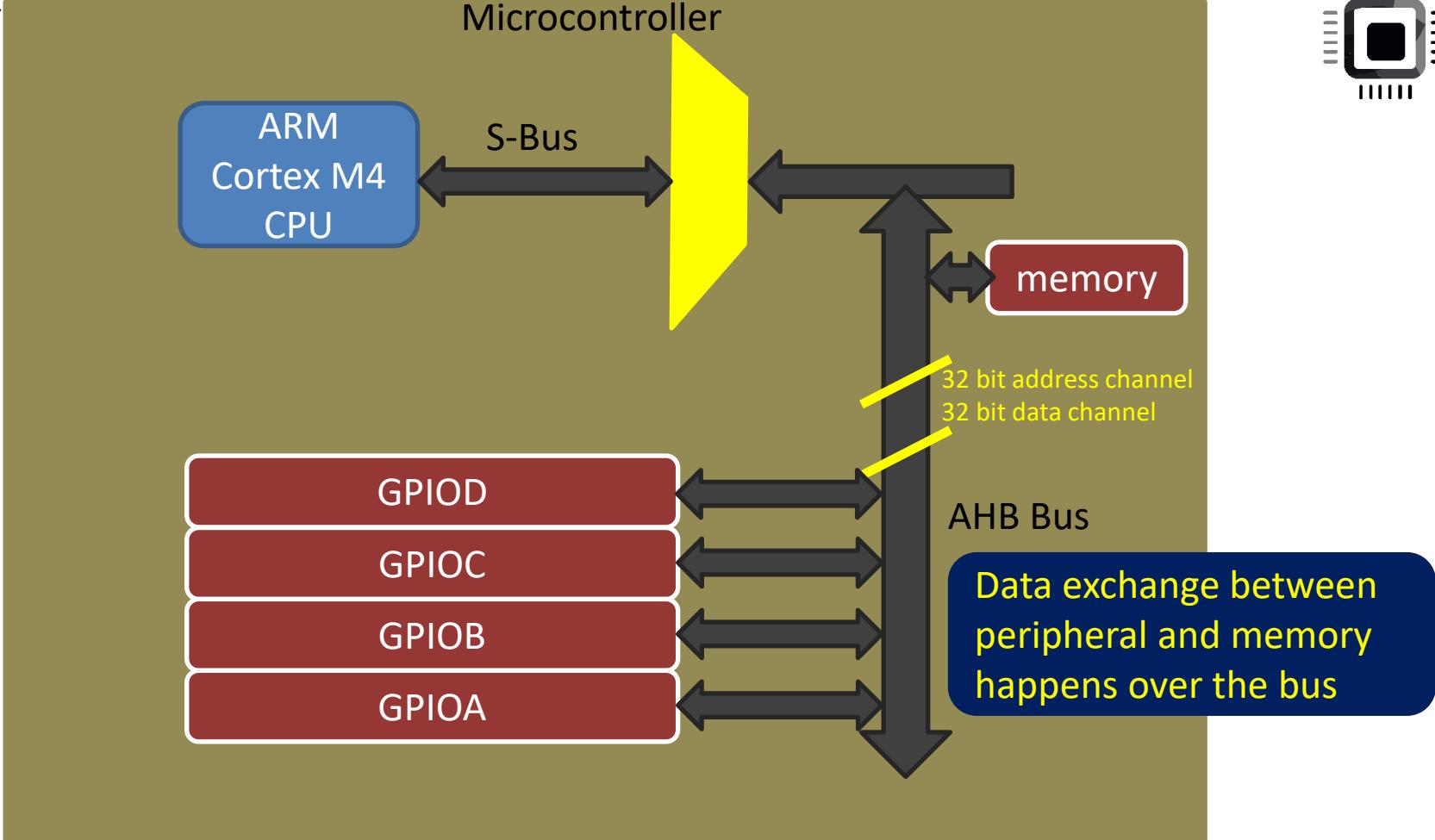
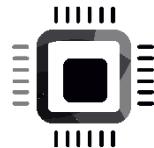
- 1) Identify the GPIO port(a peripheral) used to connect the LED
- 2) Identify the GPIO pin where the LED is connected
- 3) Activate the GPIOD peripheral (Enable the clock)
- 4) Configure the GPIO pin mode as output
- 5) Write to the GPIO pin

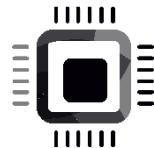


How to Enable the peripheral clock ?

1. Through peripheral clock control registers of the microcontroller
2. In STM32 MCU, all clock control registers are mapped at the below address range in the memory map of the Microcontroller.

0x4002 3800 - 0x4002 3BFF	RCC
---------------------------	-----





Microcontroller

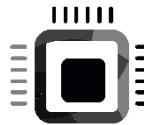
ARM
Cortex M4
CPU

S-Bus

GPIOD
GPIOC
GPIOB
GPIOA

AHB1 Bus

Peripheral and memory
exchange data over this



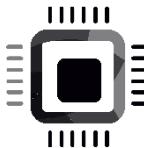
RCC AHB1 peripheral clock enable register (RCC_AHB1ENR)

Address offset: 0x30

Reset value: 0x0010 0000

Access: no wait state, word, half-word and byte access.

	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	
Reser- ved	OTGH S ULPIE N	OTGH SEN	ETHM ACPTP EN	ETHM ACRXE N	ETHM ACTXE N	ETHMA CEN	Reserved		DMA2E N	DMA1E N	CCMDAT ARAMEN	Res.	BKPSR AMEN	Reserved			
	rw	rw	rw	rw	rw	rw			rw	rw			rw				
	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
Reserved			CRCE N	Reserved			GPIOE N	GPIOH EN	GPIOG EN	GPIOFE N	GPIOEEN	GPIOD EN	GPIOC EN	GPIO BEN	GPIO AEN		
			rw				rw	rw	rw	rw	rw	rw	rw	rw			



GPIO port mode register (GPIOx_MODER) (x = A..I/J/K)

Address offset: 0x00

Reset values:

- 0xA800 0000 for port A
- 0x0000 0280 for port B
- 0x0000 0000 for other ports

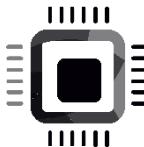
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
MODER15[1:0]		MODER14[1:0]		MODER13[1:0]		MODER12[1:0]		MODER11[1:0]		MODER10[1:0]		MODER9[1:0]		MODER8[1:0]	
rw	rw	rw	rw	rw	rw										
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
MODER7[1:0]		MODER6[1:0]		MODER5[1:0]		MODER4[1:0]		MODER3[1:0]		MODER2[1:0]		MODER1[1:0]		MODER0[1:0]	
rw	rw	rw	rw	rw	rw										

Bits 2y:2y+1 **MODERy[1:0]**: Port x configuration bits (y = 0..15)

These bits are written by software to configure the I/O direction mode.

00: Input (reset state)

01: General purpose output mode



8.4.6 GPIO port output data register (GPIOx_ODR) (x = A..I/J/K)

Address offset: 0x14

Reset value: 0x0000 0000

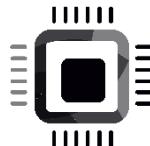
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved															
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
ODR15	ODR14	ODR13	ODR12	ODR11	ODR10	ODR9	ODR8	ODR7	ODR6	ODR5	ODR4	ODR3	ODR2	ODR1	ODR0
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

Bits 31:16 Reserved, must be kept at reset value.

Bits 15:0 **ODRy**: Port output data (y = 0..15)

These bits can be read and written by software.

Note: For atomic bit set/reset, the ODR bits can be individually set and reset by writing to the GPIOx_BSRR register (x = A..I/J/K).



Bitwise Operators in 'C'

&

(bitwise AND)

>>

(bitwise Right Shift)

|

(bitwise OR)

Unary
~

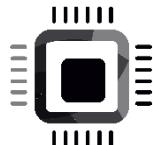
(bitwise NOT) (Negation)

<<

(bitwise Left Shift)

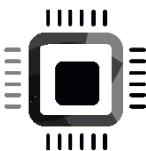
^

(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



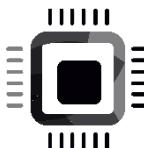
char a = 111

char b = a >> 4

b = ?

a 0 1 1 0 1 1 1 1

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



a



111 (0x6F)

$a >> 1$



$a >> 2$



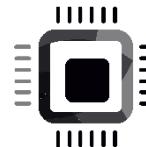
$a >> 3$



$a >> 4$



6 (0X06)



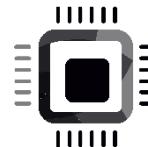
0 0 0 0

a

0 1 1 0 1 1 1 1

$a \gg 4$

0 0 0 0 0 1 1 0



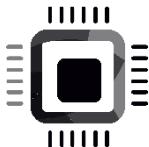
0 0 0

a

0 1 1 0 1 1 1 1

$A \gg 3$

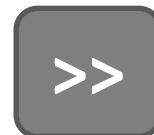
0 0 0 0 1 1 0 1



Bitwise Operators in 'C'



(bitwise AND)



(bitwise Right Shift)



(bitwise OR)



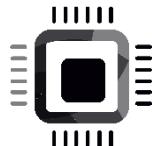
(bitwise NOT) (Negation)



(bitwise Left Shift)

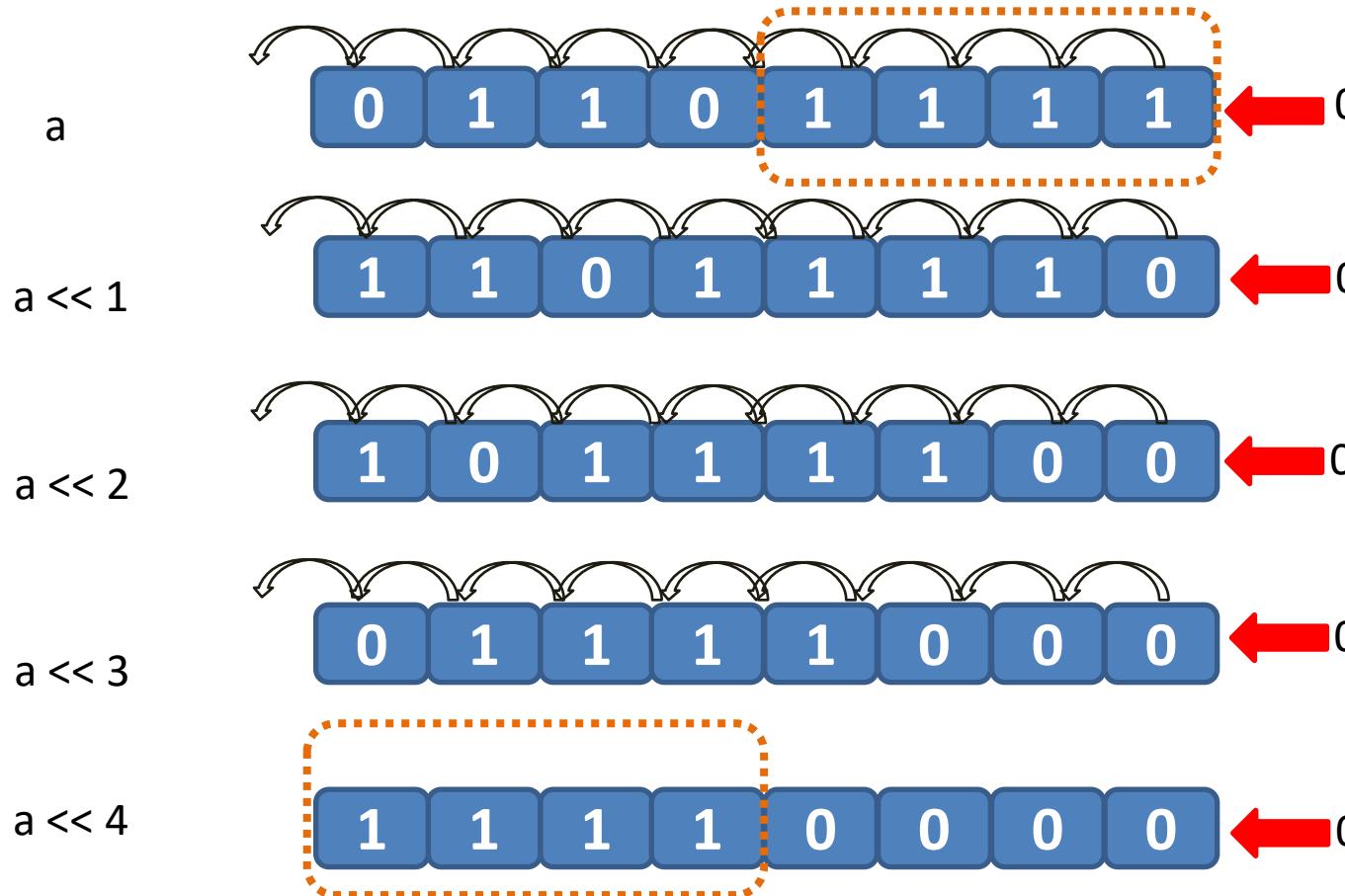
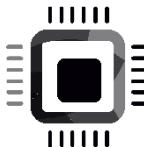


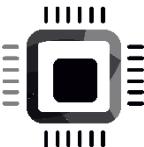
(bitwise XOR)



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

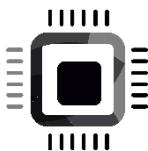




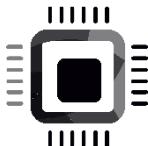
<< VS >>

<< (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 4th bit of the given data

Set 4th bit of the given data

Data = 0x08

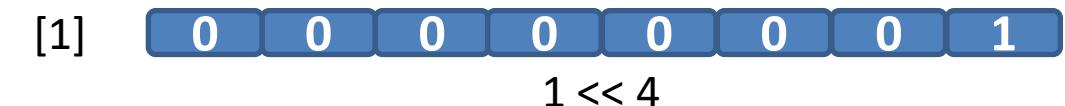
Data = Data | 0x10
= 0x18



Set 4th bit of the given data

Data = 0x108

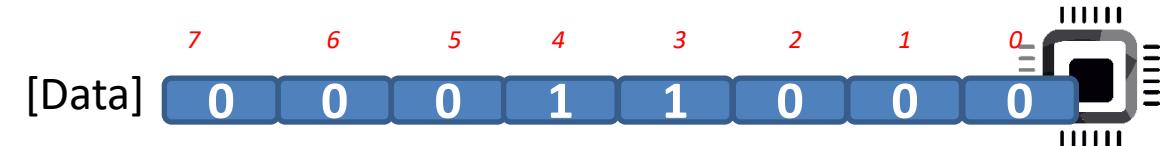
Data = Data | (1 << 4)
= 0x18



Clear 4th bit of the given data

Data = 0x18

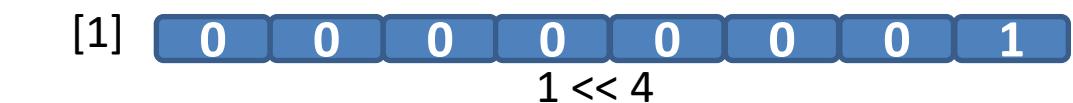
Data = Data & 0xEF
= 0x10

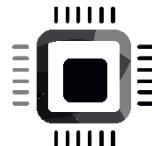


Clear 4th bit of the given data

Data = 0x18

Data = Data & ~(1 << 4)
= 0x10

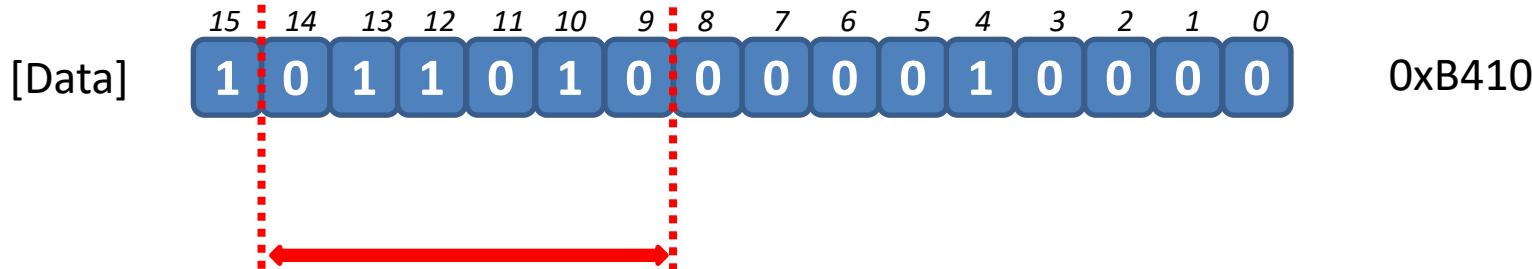
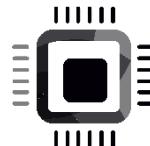




Bit extraction

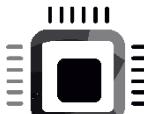
- Lets consider this problem statement

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

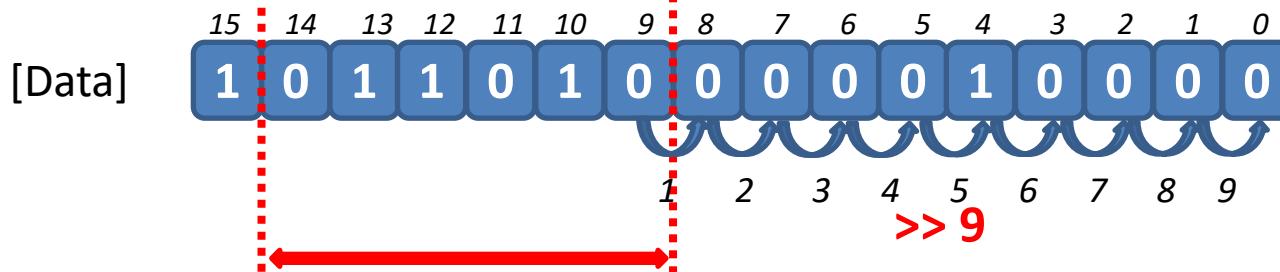


This portion needs to be extracted and saved in to another variable

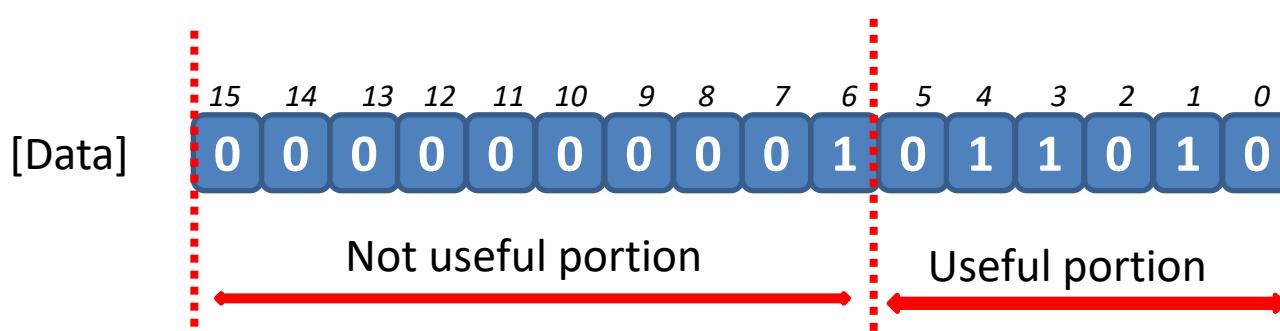
- 1) Shift the identified portion to right hand side until it touches the least significant bit (0th bit)
- 2) Mask the value to extract only 6 bits [5:0] and then save it in to another variable



Original data

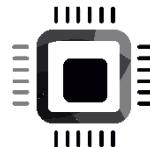


- Shift the identified portion to right hand side until it touches the least significant bit (0th bit)

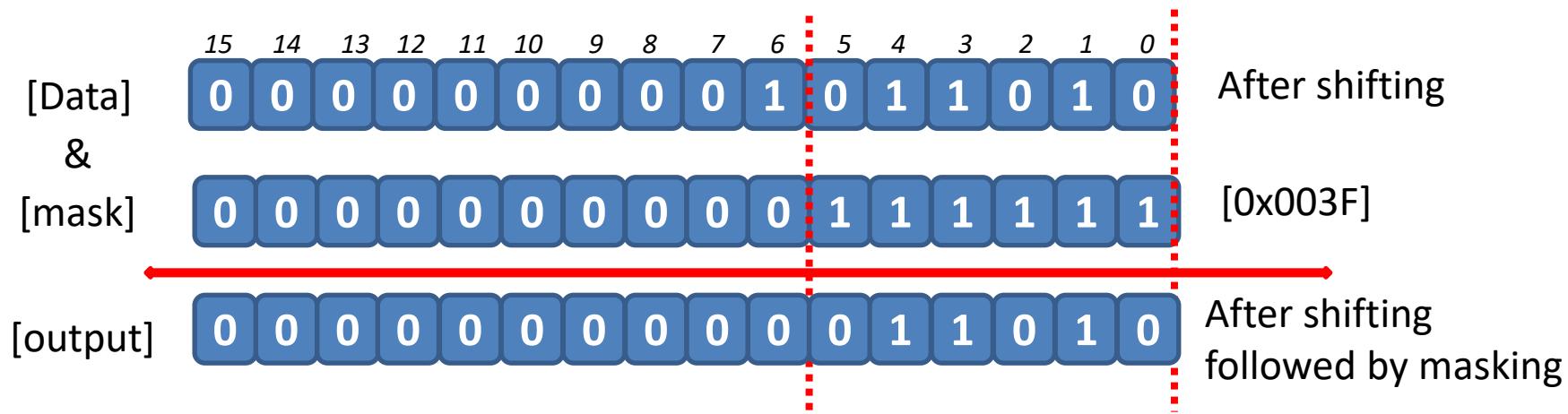


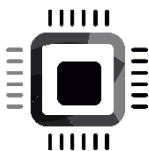
After shifting

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



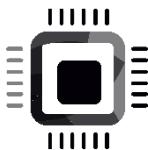
2) Mask the new value to extract only first 6 bits[5:0] and then save it in to another variable





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

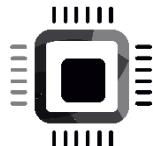
```
output = (uint8_t) ((Data >> 9) & 0x003F)
```



- Solve this problem using combinations of bitwise operators

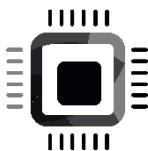
Set bit positions [10:5] in a given data to 0x3E

Data = 0x4f63;

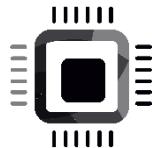


Set bit positions [10:5] in a given data to 0x3E

Data = 0x4f63;

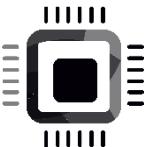


Decision Making in ‘C’



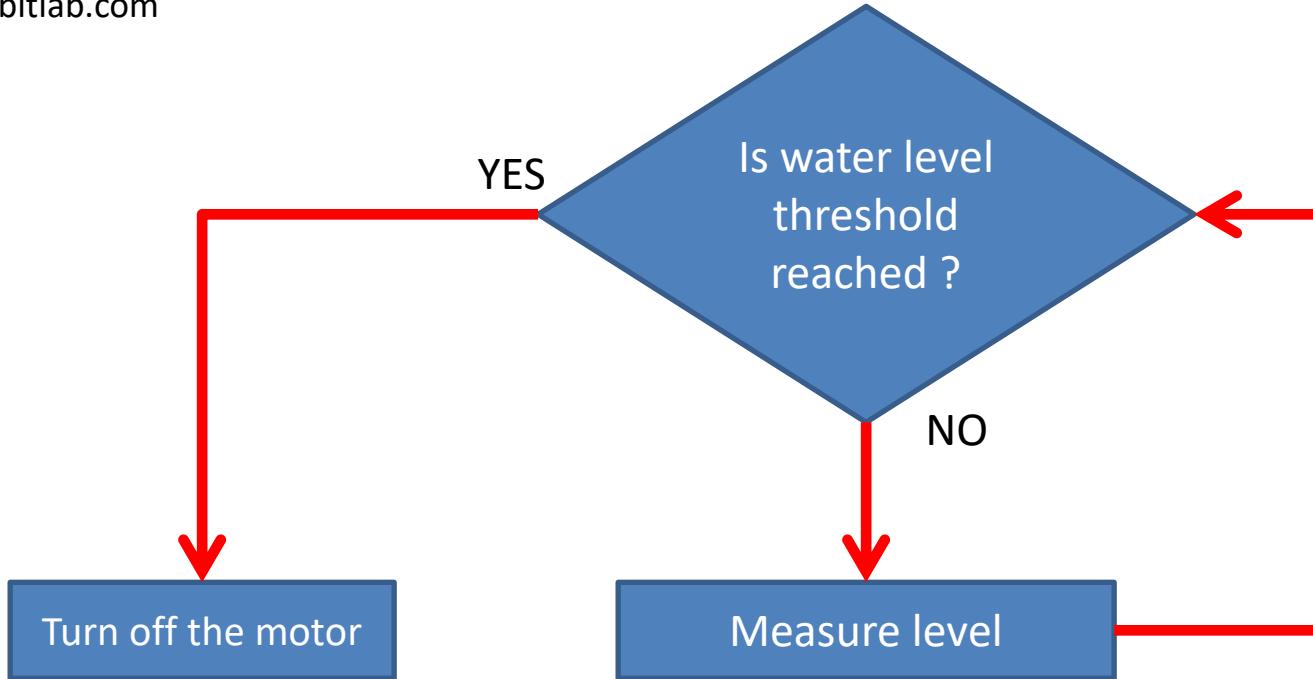
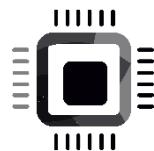
Decision Making in ‘C’

- ✓ When you are going to write some program, it will always be necessary that you will have to put some code which has to be executed only when a specific condition is satisfied
- ✓ Your program has to take decision based on internal/external events or conditions.

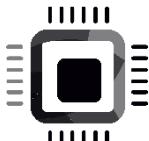


Decision Making in 'C'

- ✓ For example, if the user presses a key, then execute 'this' set of statements or if the user doesn't press any key, then execute 'that' set of statements.
- ✓ In embedded programming, let's take an example of water level indication and control program. If the sensor detects water level rising above the threshold, then the program executes the code to turn off the motor otherwise program doesn't turn off the motor.



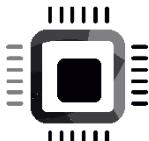
Decision making in 'C'



Decision taking statements in 'C'

In 'C' there are 5 different ways to take decisions by making use of below decision taking statements .

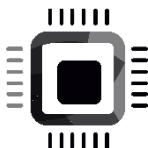
1. if Statement
2. if-else Statement
3. if-else-if ladder
4. Conditional Operators
5. Switch/case Statement



Decision taking statements in 'C'

In 'C' there are 5 different ways to take decisions by making use of below decision taking statements .

1. if Statement
2. if-else Statement
3. if-else-if ladder
4. Conditional Operators
5. Switch/case Statement



Syntax : if statement

Single statement execution

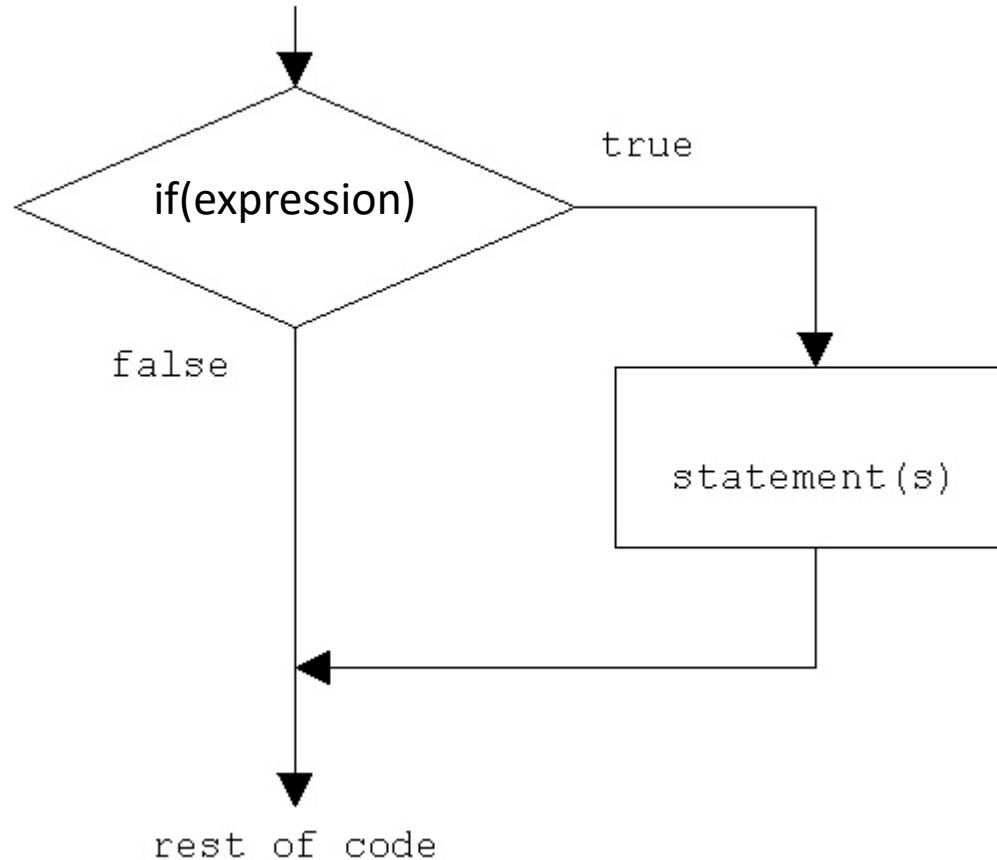
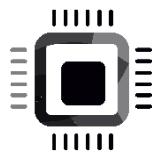
```
if(expression)  
    statement;
```

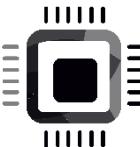
If the expression evaluation is true
(nonzero), then only statement(s)
will be executed

Multiple statement execution

```
if(expression) {  
    statement_1;  
    statement_2;  
    statement_3;  
    statement_4;  
    statement_n;  
}
```

if statement flow chart





```
int main(void)
{
    uint8_t myData = 20;

    if (myData > 40)
        printf("Value = %d\n",myData);

    /* this statement is outside if block , so always executed */
    myData++;

    return 0;
}

int main(void)
{
    uint8_t myData = 60;

    if (myData > 40)
    {
        printf("Value = %d\n",myData);
        myData=0;
    }

    /* this statement is outside if block , so always executed */
    myData++;

    return 0;
}
```

```
int main(void)
{
    uint8_t myData = 20;
```

```
if (myData > 40)
    printf("Value = %d\n",myData);
```

```
/* this statement is outside if block , so always executed */
myData++;
```

```
return 0;
```

```
}
```

```
int main(void)
{
    uint8_t myData = 60;
```

```
if (myData > 40)
{
    printf("Value = %d\n",myData);
    myData=0;
}
```

```
/* this statement is outside if block , so always executed */
myData++;
```

```
return 0;
```

```
}
```

This is called a “conditional statement(s).”
Because its execution depends upon evaluation
of an expression (myData > 40)

```
int main(void)
{
    uint8_t myData = 20;

    if (myData > 40)
        printf("Value = %d\n",myData);
}
```

Beginner's mistake :
Observe that no ; here

```
/* this statement is outside if block , so always executed */
myData++;

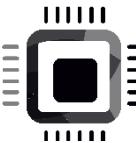
return 0;
```

```
int main(void)
{
    uint8_t myData = 60;

    if (myData > 40)
    {
        printf("Value = %d\n",myData);
        myData=0;
    }

    /* this statement is outside if block , so always executed */
    myData++;

    return 0;
}
```



In C , a semicolon (;) by itself or an empty block ({}) is an NOP

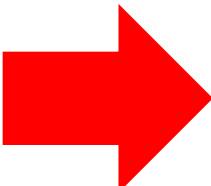
```
int main(void)
{
    uint8_t myData = 60;

    if (myData > 40)
    {
        printf("Value = %d\n",myData);
        myData=0;
    }

    /* this statement is outside if block */
    myData++;

    return 0;
}
```

Equivalent to



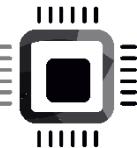
```
int main(void)
{
    uint8_t myData = 60;

    if (myData > 40)
    {
        ;
    }

    {
        printf("Value = %d\n",myData);
        myData=0;
    }

    /* this statement is outside if block */
    myData++;

    return 0;
}
```



```
int main(void)
{
    uint8_t myData = 20;

    if (myData > 40)
        printf("Value = %d\n",myData);

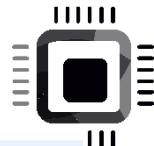
    /* this statement is outside if block , so always executed */
}
```

The relational operators (`==`, `>`, `<`, `>=`, `<=`, `!=`) and the logical operators (`&&`, `||`, `!`) are frequently used as the basis of conditions in *if and if/else conditional statements to direct program flow*.

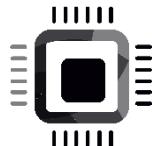
```
    printf("Value = %d\n",myData),
    myData=0;
}

/* this statement is outside if block , so always executed */
myData++;

return 0;
}
```

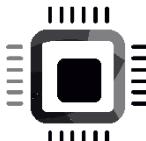


```
uint8_t isButtonPressed = 0;  
  
int main(void)  
{  
  
    if (isButtonPressed)  
    {  
        printf("Turn on the LED\n");  
        isButtonPressed = 0;  
    }  
  
    return 0;  
  
}  
  
/* Interrupt handler for button press */  
void ISR_button(void)  
{  
    isButtonPressed = 1;  
}
```



Exercise

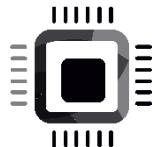
- Write a program that takes the user's age and decides whether a user can cast a vote or not.
- The minimum age for casting a vote is 18 years
- Print appropriate messages



Decision taking statements in 'C'

In 'C' there are 5 different ways to take decisions by making use of below decision taking statements .

1. if Statement
2. **if-else Statement**
3. if-else-if ladder
4. Conditional Operators
5. Switch/case Statement



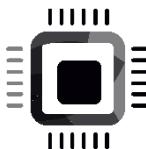
Syntax : if...else....statement

Single statement execution

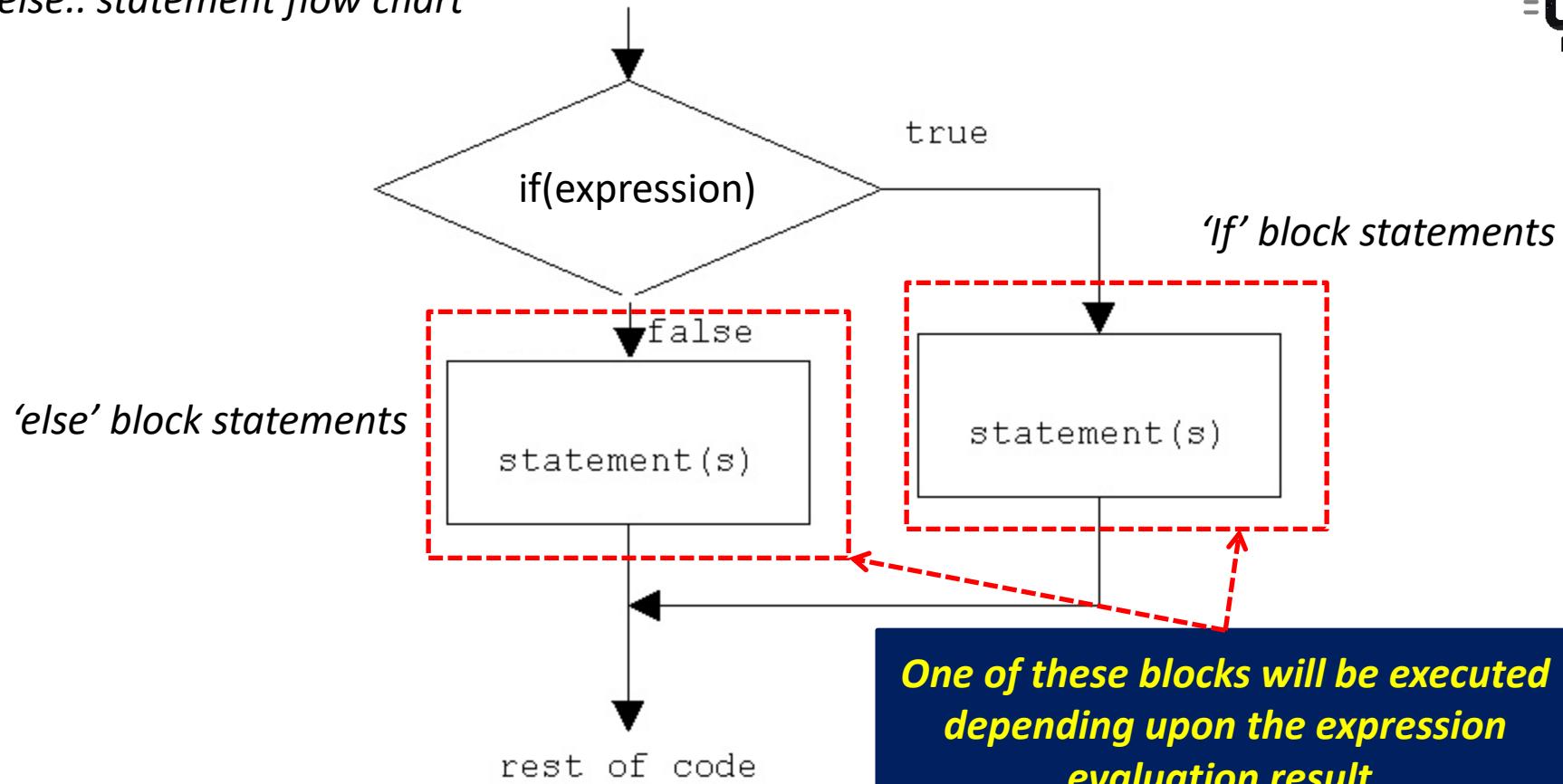
```
if(expression)
    statement_1;
else
    statement_2;
```

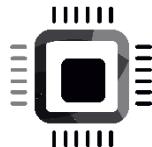
Multiple statement execution

```
if (expression)
{
    statement_1;
    statement_2;
}
else
{
    statement_3;
    statement_4;
}
```



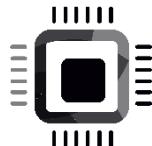
If..else.. statement flow chart





Exercise

- Write a program which receives 2 numbers (integers) from the user and prints the biggest of two .
- If $n1 == n2$, then print “both numbers are equal”



Return value of scanf

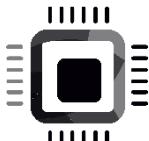
Here, **scanf** returns total number of inputs scanned successfully

```
scanf("%f", &num1);
```

In this above case **scanf** returns 1 , if the scan is successful.

If the scan is unsuccessful then **scanf** returns 0.

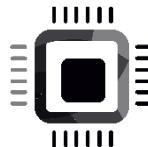
So, when you enter a character for the above code , the **scanf** fails and returns 0.
The **scanf** fails for the character input because it was expecting a number



Decision taking statements in 'C'

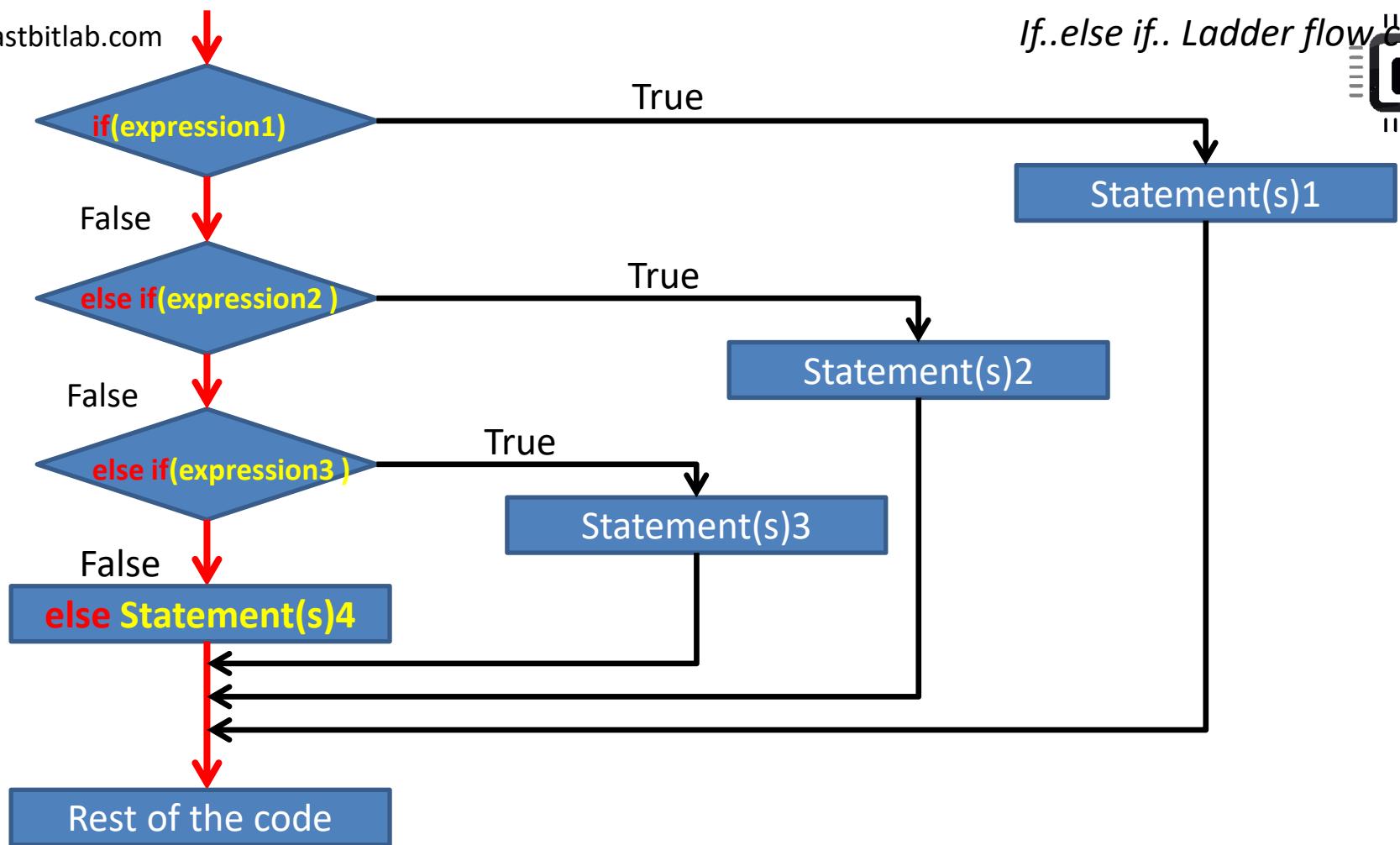
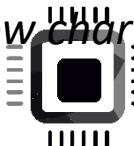
In 'C' there are 5 different ways to take decisions by making use of below decision taking statements .

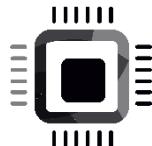
1. if Statement
2. if-else Statement
3. **if-else-if ladder**
4. Conditional Operators
5. Switch/case Statement



Syntax : if..else.. ladder statement

```
if (expression1) ← First check this expression
{
    // statement(s)
}
else if (expression2) ← Check this expression only if above expression
{
    // statement(s)
}
else if (expression3) ← Check this expression only if above expression is
{
    // statement(s)
}
.
.
else
{
    // statement(s) ← Execute these statements only if all the above
    // expression checks are false.
}
```



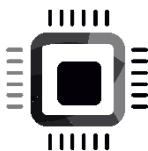


Exercise

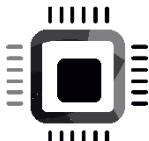
- Write a program to calculate income tax payable of the user . Tax is calculate as per below table.

Total income	% of tax
Up to \$9,525	0
\$9,526 to \$38,700	12%
\$38,701 to \$82,500	22%
> \$82,500	32% + \$1000

Tax payable = Income * (tax rate / 100)



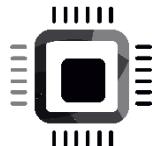
```
else if( (income > 9525) && ( income <= 38700 ) )
{
    tax = income * 0.12;
}
```



Decision taking statements in 'C'

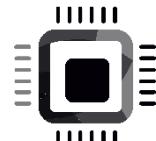
In 'C' there are 5 different ways to take decisions by making use of below decision taking statements .

1. if Statement
2. if-else Statement
3. if-else-if ladder
4. **Conditional Operators**
5. Switch/case Statement



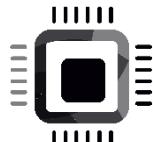
Conditional Operators

- Conditional operator is a ternary operator in C used for conditional evaluation.
- Operator symbol **?:**
- It's a ternary operator because it operates on three operands.



Syntax of conditional operator

expression1 ? expression2 : expression3



Syntax of conditional operator

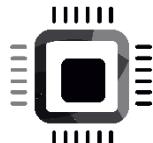
expression1 ? expression2 : expression3

(operand 1)

(operand 2)

(operand 3)

```
uint32_t a = ( 5 + 4 ) ? (9 -4 ) : 99;
```



Syntax of conditional operator

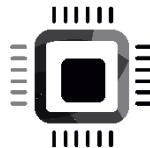
expression1 ? expression2 : expression3

(operand 1)

(operand 2)

(operand 3)

```
uint32_t a = ( 5 + 4 ) ? (9 -4 ) : 99;  
a = 5;
```



TRUE

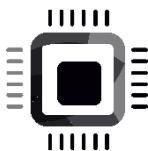
expression1 ? expression2 : expression3

This will be the final result

FALSE

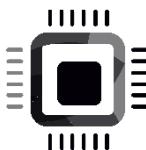
expression1 ? expression2 : expression3

This will be the final result



Use case

- Sometimes you can use conditional operator to replace *if..else* statements

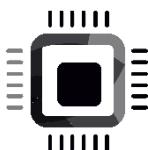


Using if....else

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

    if( age < 18)
    {
        printf("you are not eligible to vote\n");
    }else
    {
        printf(" Congrats !!! you can vote\n");
    }

    return 0;
}
```

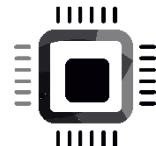


Using Conditional operator

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

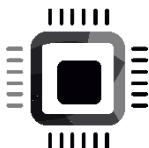
    (age < 18) ? printf("you are not eligible to vote\n") : printf(" Congrats !!! you can vote\n");

    return 0;
}
```



Evaluate these expressions

```
int32_t a = 5;  
a = 0?(a < 9) : a++;  
a = ?
```

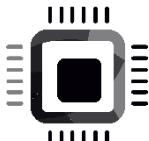


Evaluate these expressions

```
int32_t a = 5, b = 10;
```

```
a = !(a+b) ? !(a < 9) : a ;
```

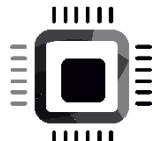
```
a = ?
```



Decision taking statements in 'C'

In 'C' there are 5 different ways to take decisions by making use of below decision taking statements .

1. if Statement
2. if-else Statement
3. if-else-if ladder
4. Conditional Operators
5. **Switch/case Statement**



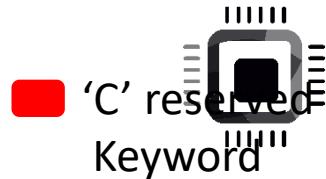
Syntax of switch/case statement

- Switch/Case statement is one of the decision making statement available in ‘C’
- It may be used instead of several if...else statements.

Syntax of switch/case statement

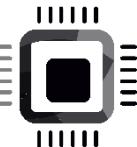
```
switch(expression )  
{  
    case value :  
        Statement-1;  
        Statement-n;  
        break;  
    case value :  
        Statement-1;  
        Statement-n;  
        break;  
    case value :  
        Statement-1;  
        Statement-n;  
        break;  
    default:  
        Statement-1;  
        Statement-n;  
}
```

■ ‘C’ reserved
Keyword



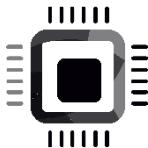
```
switch(expression )  
{  
    case value :  
        Statement-1,  
        Statement-n;  
        break;  
    case value :  
        Statement-1;  
        Statement-n;  
        break;  
    case value :  
        Statement-1;  
        Statement-n;  
        break;  
    default:  
        Statement-1;  
        Statement-n;  
}
```

Case label and it must be
an integer value



```
www.fastbith.com
int main(void)
{
    uint8_t key_read = read_keypad();

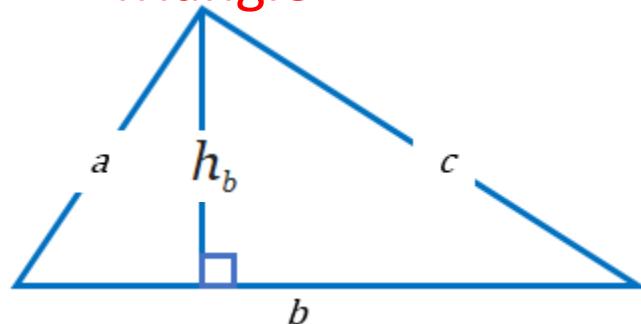
    switch(key_read)
    {
        case 1:
            all_leds_race();
            break;
        case 2:
            all_leds_on();
            break;
        case 3:
            all_leds_toggle();
            break;
        case 4:
            all_leds_blink();
            break;
        default :
            all_leds_off();
            printf("Invalid key ! Please enter number between (1 to 4) only\n");
    }
}
```



Exercise

- Write a program to calculate the area of different geometrical figures
 - Circle , triangle , trapezoid , square and rectangle
- The program should ask the user to enter the code for which user wants to find out the area
 - ‘t’ for triangle
 - ‘z’ trapezoid
 - ‘c’ circle
 - ‘s’ square
 - ‘r’ rectangle

Triangle

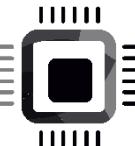


$$A = \frac{h_b b}{2}$$

b Base

h_b Height

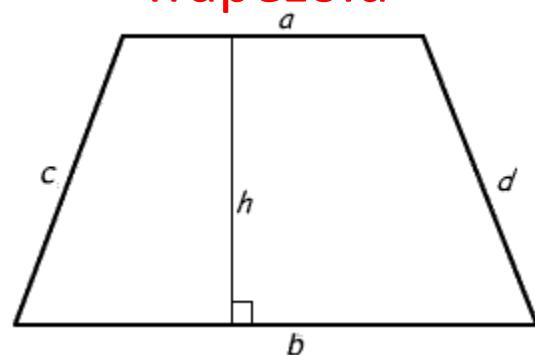
Circle



$$A = \pi r^2$$

r Radius

Trapezoid



$$A = \frac{a+b}{2} h$$

a Base

b Base

h Height

a

a

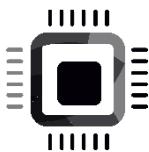
Square

a

a

$$A = a^2$$

a Side

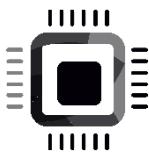


$$A = w l$$

l Length

w Width

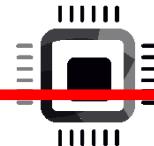
length



Looping in ‘C’

Without loop

Program which prints numbers from 1 to 10

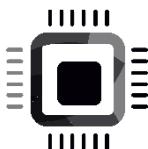
With loop

```
#include<stdio.h>

int main(void)
{
    printf("1\n");
    printf("2\n");
    printf("3\n");
    printf("4\n");
    printf("5\n");
    printf("6\n");
    printf("7\n");
    printf("8\n");
    printf("9\n");
    printf("10\n");
}
```

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

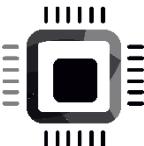
int main(void)
{
    uint8_t i = 1;
    while( i <= 10 )
    {
        printf("%d\n",i++);
    }
}
```



Loops in 'C'

- There are three types of loop statements in 'C'
 - *while* loop
 - *for* loop
 - do...while loop

**while, for, do
('C' reserved keywords)**



Syntax : while loop

Loop body start

while(expression)

{ /* multi statement while loop */
statement1;

statement2;

statement3;

statementN;

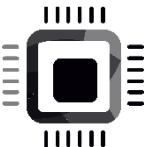
}

Loop body end

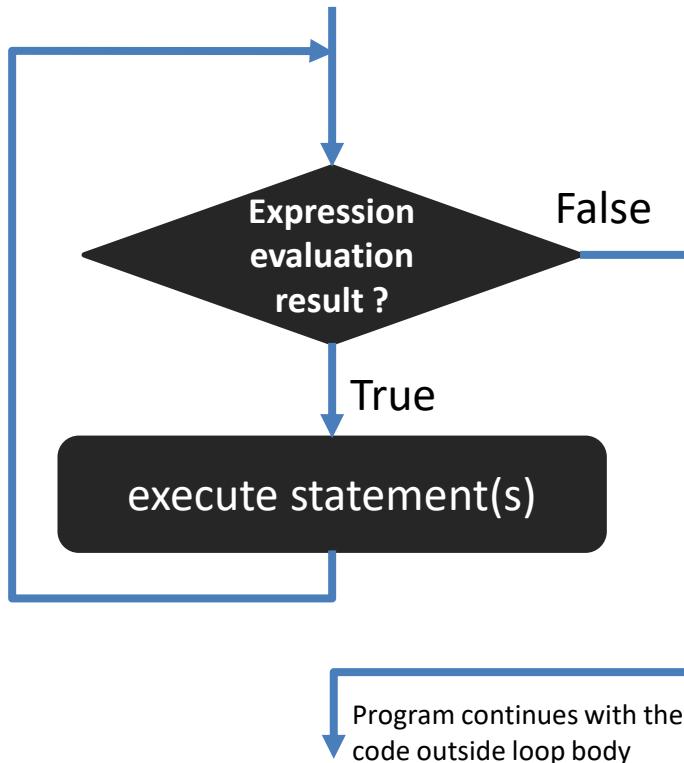
/*Single statement while loop*/

while(expression)
statement1;

Repeat execution of code inside the loop
body until expression evaluates to false(0)

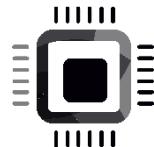


Flowchart of while loop



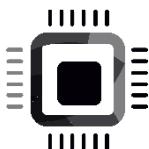
1. expression is evaluated first
2. If expression evaluation is TRUE, then statements inside the body of the loop will be executed and execution loops back to check the expression again.
3. If expression evaluation is FALSE, loop body breaks and the program continues with code outside the loop body

In while loop expression is always evaluated first



Example

- Write a program which prints from 1 to 10 using ‘while’ loop



while loop and ;

```
int main(void)
{
    uint8_t i = 1;

    while(i <= 10);
    {
        printf("%d\n",i++);
    }

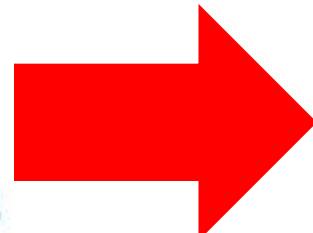
}
```

while loop and semicolon

In C , a semicolon (;) by itself or an empty block ({}) is an NOP

```
int main(void)
{
    uint8_t i = 1;
    while(i <= 10)
    {
        printf("%d\n",i++);
    }
}
```

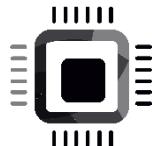
Equivalent to



```
int main(void)
{
    uint8_t i = 1;

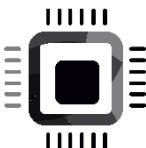
    while(i <= 10)
    {
        ; /*do nothing (NOP) */
    }

    {
        printf("%d\n",i++);
    }
}
```



Example

- Write a program to print all even numbers between 0 to 100(including boundaries)
- Also count and print how many even numbers you find.
 - Use ‘while’ loop or ‘for’ loop or ‘do while’ loop
 - Later change the program to accept boundary numbers from the user

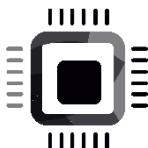


Hangs forever

```
int main(void)
{
    uint8_t i = 1;

    while(i <= 10)
    {
        printf("%d\n",i++);
    }

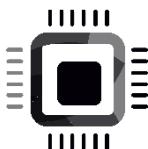
    //Code hangs forever here. main never returns
    //Used most of the time in microcontroller programming using 'C'
    while(1);
}
```



forever loop

A special form of the while loop is the forever loop. This is a loop that never ends. It is common to see this in an embedded application in the main program. Unlike a PC program, an embedded program may just run forever (or as long as it is powered up).

```
int main(void)
{
    /*super loop */
    while(1)
    {
        readDataSensor();
        processData();
        dispalySensorData();
    }
}
```

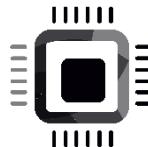


Loops in 'C'

- There are three types of loop statements in 'C'
 - *while* loop
 - *for* loop
 - do...while loop

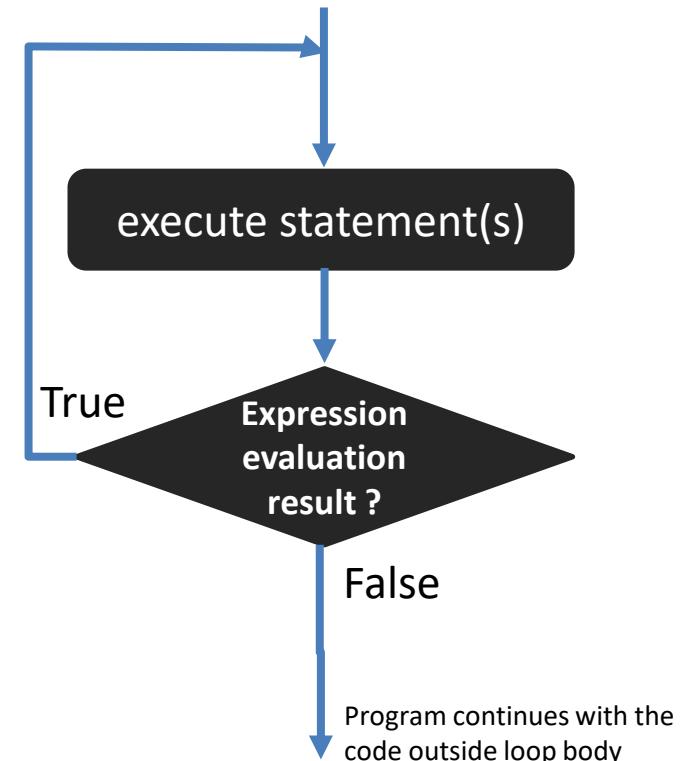
While, for, do
(standard 'C' keywords)

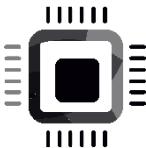
Syntax : do..while loop



```
do  
{  
    statement1  
    statement2  
    statement3  
    statementN  
}while(expression);
```

Repeat execution of code inside the “do” body until expression evaluates to false(0)





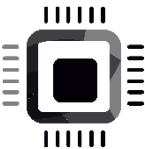
do while loop

```
do  
{  
    statement1  
    statement2  
    statement3  
    statementN  
}while(expression);
```

1. Statements are executed first
2. Expression is evaluated
3. If expression evaluation result is true,
execution loops back and the statements
inside the “do” body executes again
4. If expression evaluation result is false, while
statement is terminated and program
continues with the statement next to while
statement
5. Statements are executed at least once

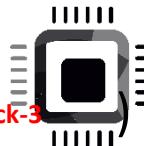
In do while loop, statement(s) is
executed first

for loop

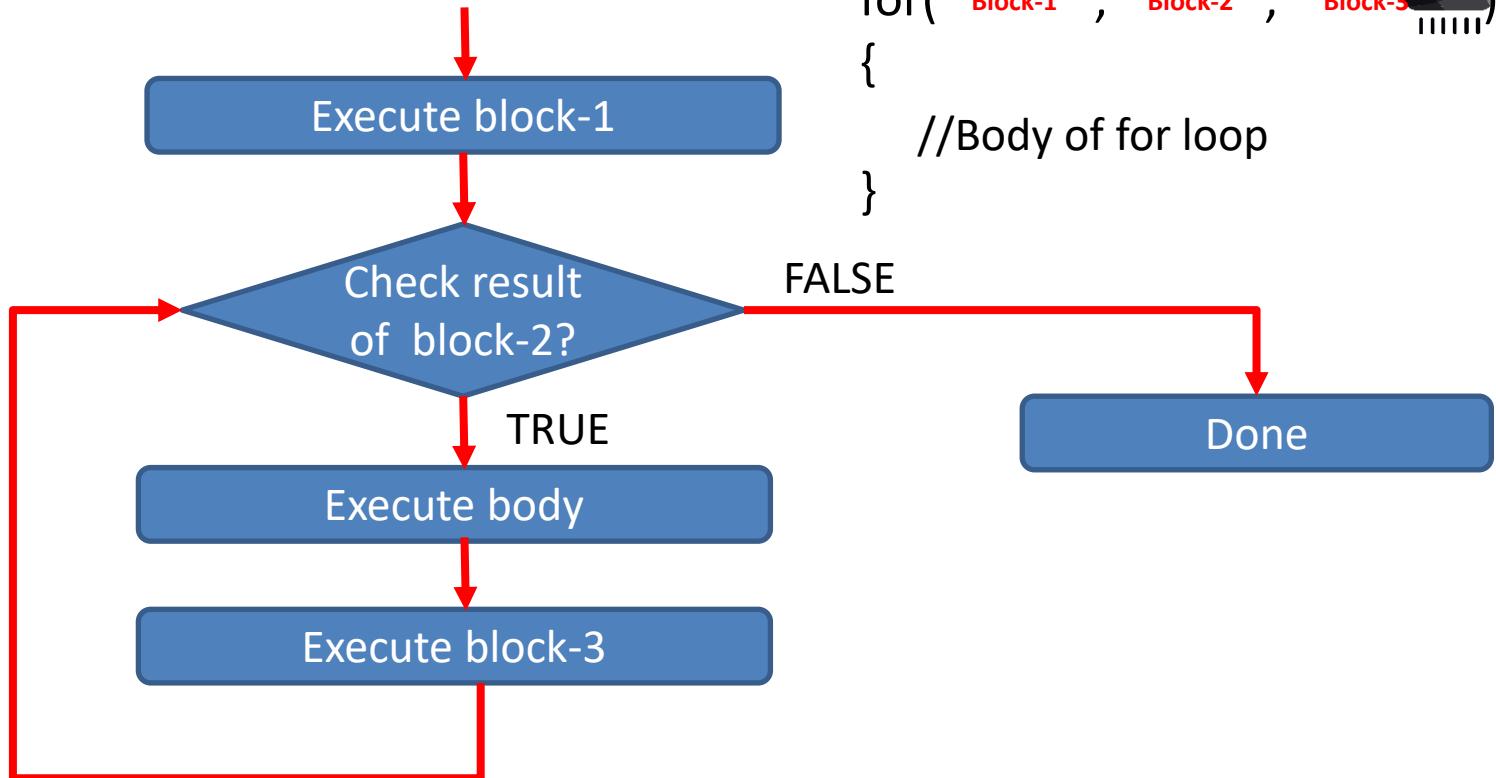


```
for(  Block-1      ;  Block-2      ;  Block-3      )  
{  
    //Body of for loop  
}
```

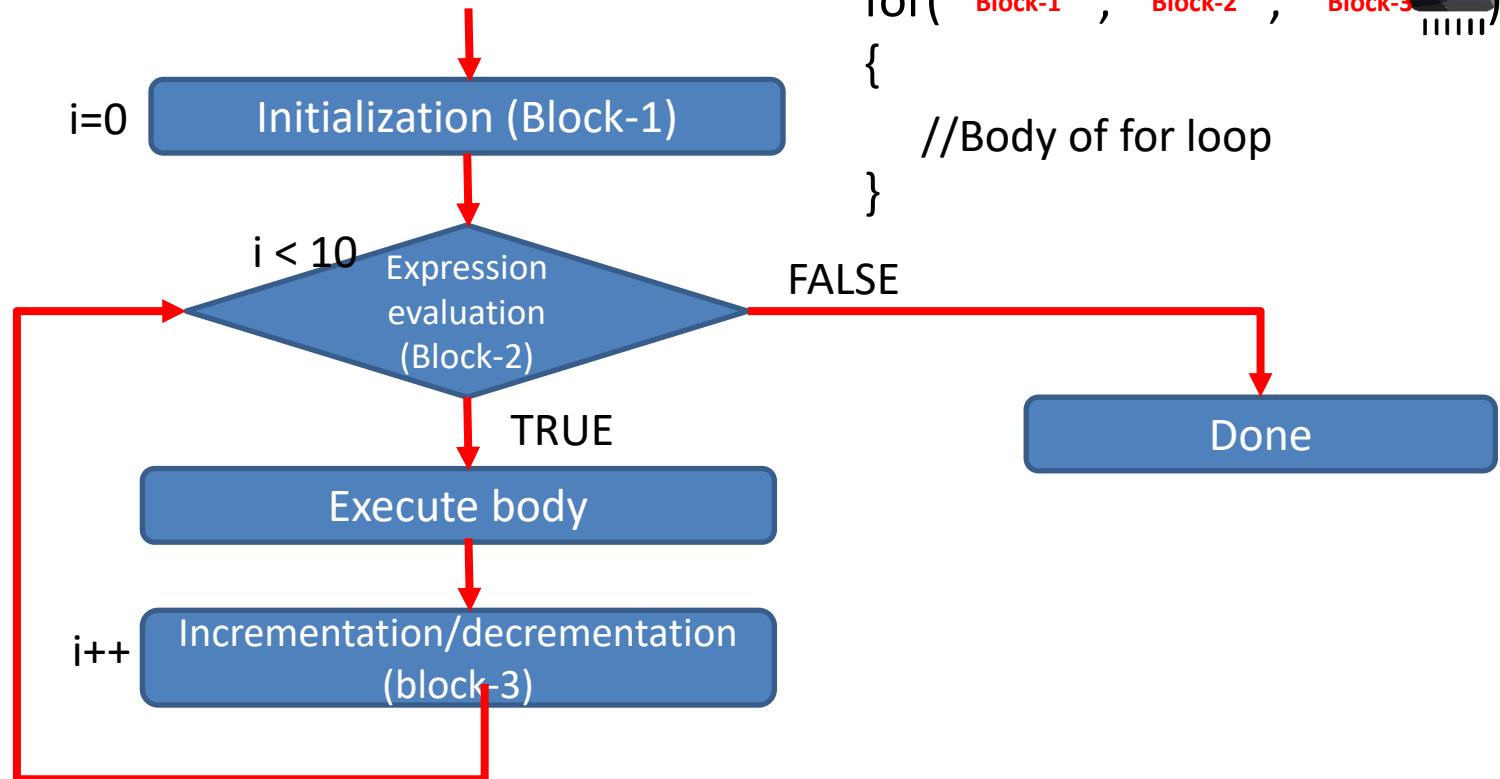
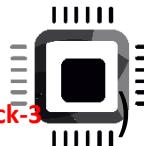
Flow chart of for loop

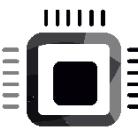


```
for( Block-1 ; Block-2 ; Block-3 )  
{  
    //Body of for loop  
}
```



Flow chart of for loop

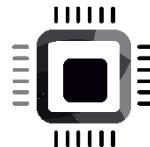




```
for( ; ; )
{
    for( ; ; )
    {
        }
    }
}
```

F:\fastbit\courses\c\repos\embedded-c\workspace_1.0.1\forloopNumberPyramid\Debug\forloopNum

```
Enter height of pyramid:10
1
2   1
3   2   1
4   3   2   1
5   4   3   2   1
6   5   4   3   2   1
7   6   5   4   3   2   1
8   7   6   5   4   3   2   1
9   8   7   6   5   4   3   2   1
10  9   8   7   6   5   4   3   2   1
Press enter key to exit this application.
```

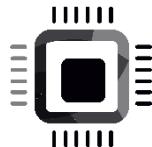


Track the height
using outer for
loop ($i < \text{height}$)

Enter height of pyramid:10

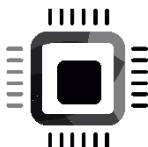
1
2 1
3 2 1
4 3 2 1 Print the rows using inner for loop(j)
5 4 3 2 1
6 5 4 3 2 1
7 6 5 4 3 2 1
8 7 6 5 4 3 2 1
9 8 7 6 5 4 3 2 1
10 9 8 7 6 5 4 3 2 1

Press enter key to exit this application.



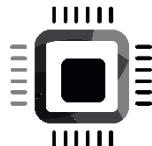
Exercise

- Modify the LED on program in to LED toggle program by creating a software delay between LED on and LED off
- LED should continuously toggle with certain time delay forever.



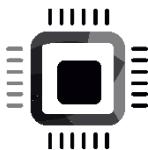
Introducing delay in program

- Software delay
 - Introduced using loop statements
- Hardware delay
 - Introduced using hardware peripherals such as timers



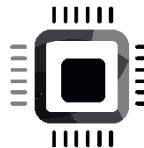
Software time delay

- To introduce software delay, you should make the processor busy in a software loop doing nothing
- This method is not an accurate method to obtain the desired time delay
- The processor will keep wasting the execution cycles until a certain delay is met

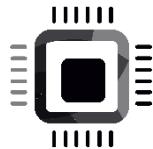


Exercise

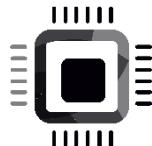
- Write a program which reads the status of the pin PA0. If the status of PA0 is LOW then turn off the on board LED(PD12) and if the status of PA0 is HIGH then turn on the LED.
- Change the status of PA0 pin manually by connecting between GND and VDD points of the board.



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

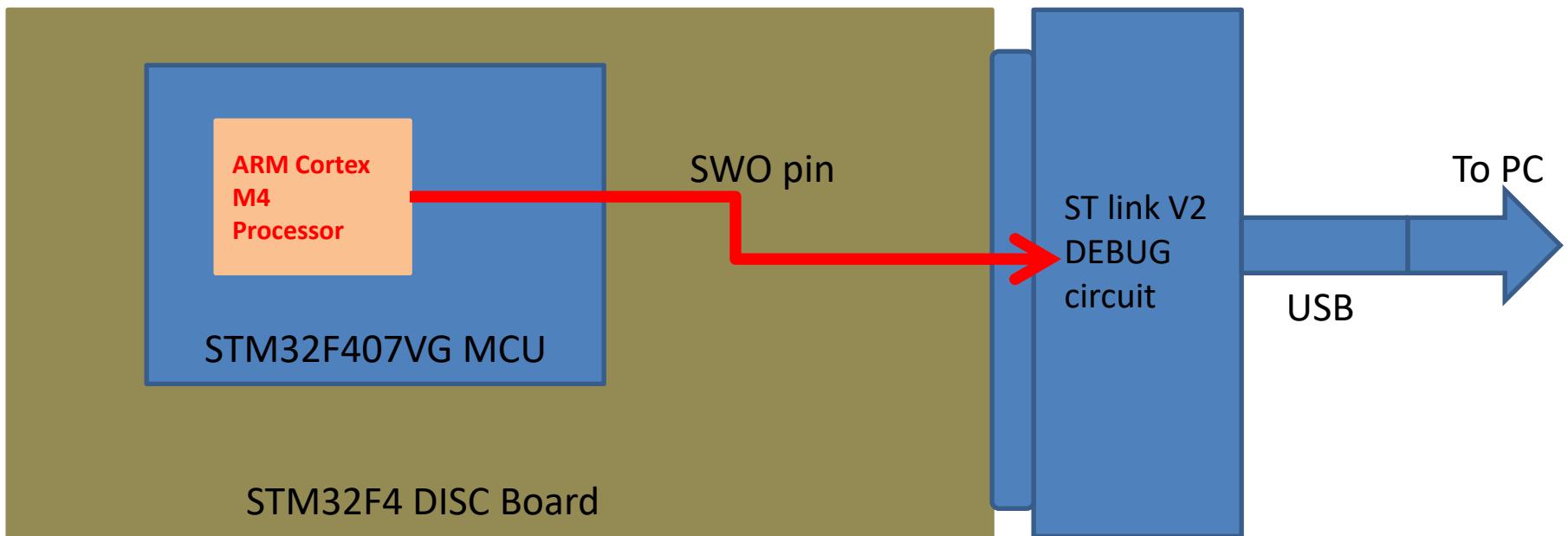
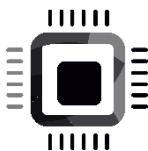


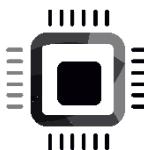
- This discussion is only applicable to MCUs based on ARM Cortex M3/M4/M7 or higher processors.
- printf works over SWO pin(Serial Wire Output) of SWD interface



Some hints

- PA0 should be in input mode
- To read from pin PA0, your code has to read PORT-A input data register





ARM Cortex M4 Processor

ITM unit

Debug connector(SWD)

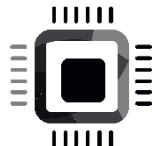
SWD(Serial Wire Debug)
2 pin (debug) + 1 pin (Trace)

Instrumentation Trace Macrocell Unit

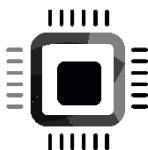
The ITM is an optional application-driven trace source that supports printf style debugging to trace operating system and application events, and generates diagnostic system information

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

SWD

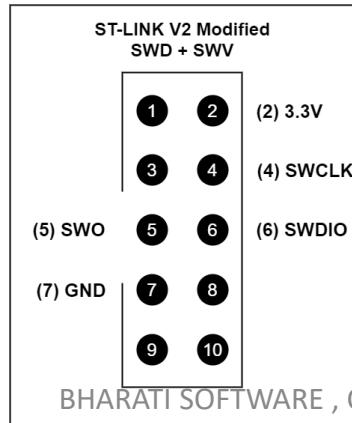


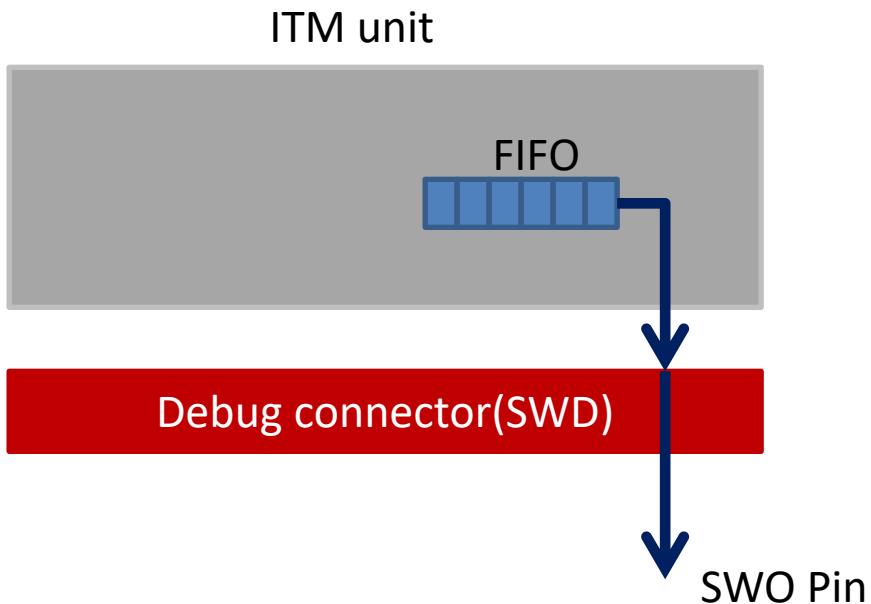
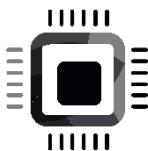
- ✓ Serial Wire Debug (SWD) is a two-wire protocol for accessing the ARM debug interface
- ✓ It is part of the ARM Debug Interface Specification v5 and is an alternative to JTAG
- ✓ The physical layer of SWD consists of two lines
 - SWDIO: a bidirectional data line
 - SWCLK: a clock driven by the host
- ✓ By using SWD interface 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.



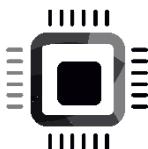
SWD and JTAG

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



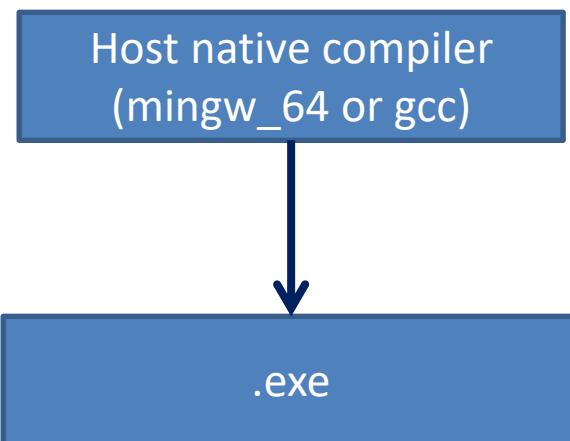


This SWO pin is connected to ST link circuitry of the board and can be captured using our debug software (IDE)



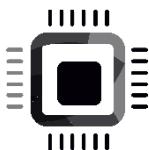
Native compilation

Host Machine



This compiler runs on host machine

Produces executable which also runs on the same machine



Cross compilation

X86/x86_64 architecture

Host Machine



ARM Architecture

Target machine

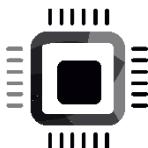


Produces executable for different
architecture

Cross compiler
arm-none-eabi-gcc

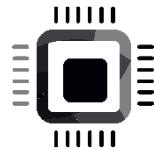
This compiler runs on host machine

.elf / .bin / .hex

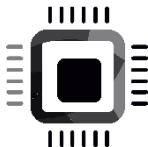


sizeof

- Find out storage sizes of the below data types for your cross compiler
 - char
 - int
 - long
 - long long
 - double



Compiler settings of STM32CubeIDE



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



preprocessor

parser

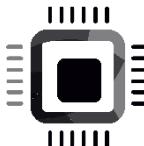
Code generator

Converts mnemonics to machine codes

Assembler

.o

Linking stage of the build



Re-locatable
object files of
your source files



Linker

Debug file



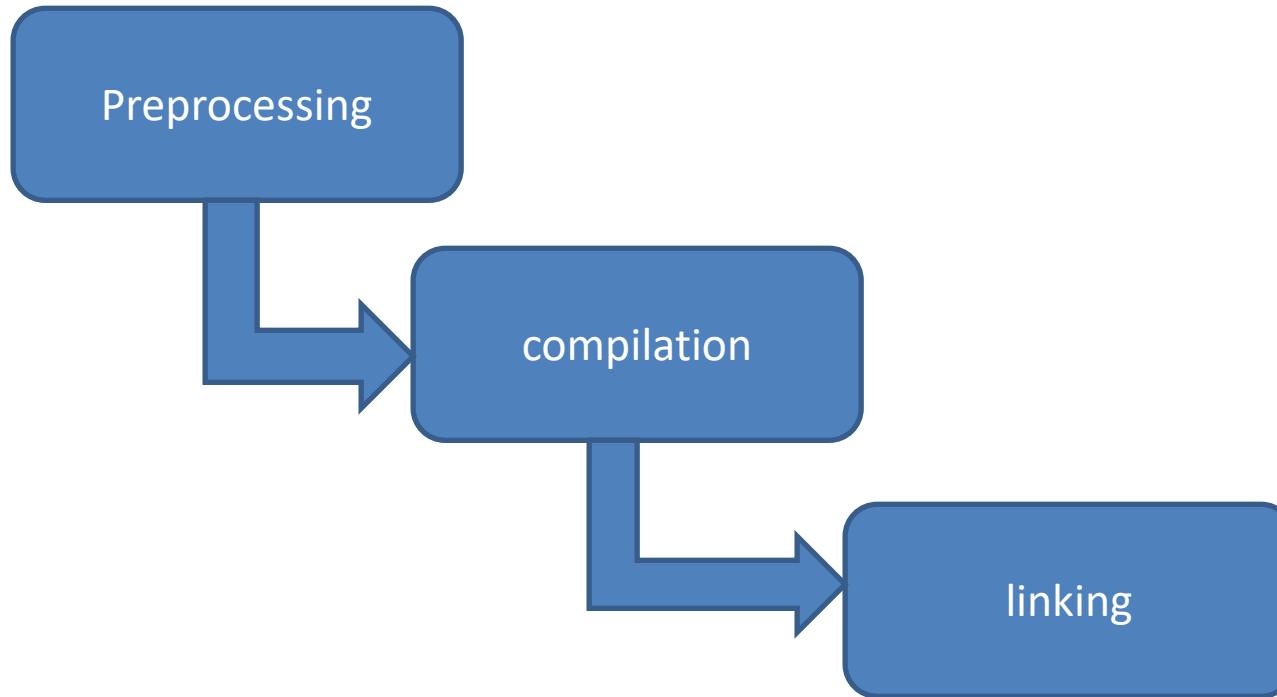
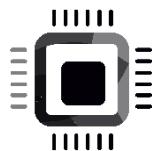
objcopy tool

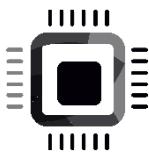


Other libraries
(std and/or third party . Ex. libc)

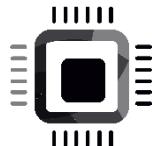
Pure binary executable file

Summary of build process



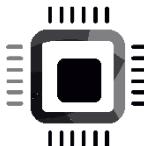
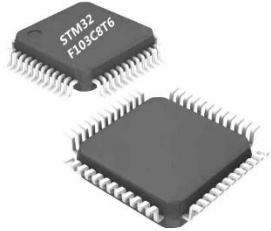


Analyzing Embedded C Program



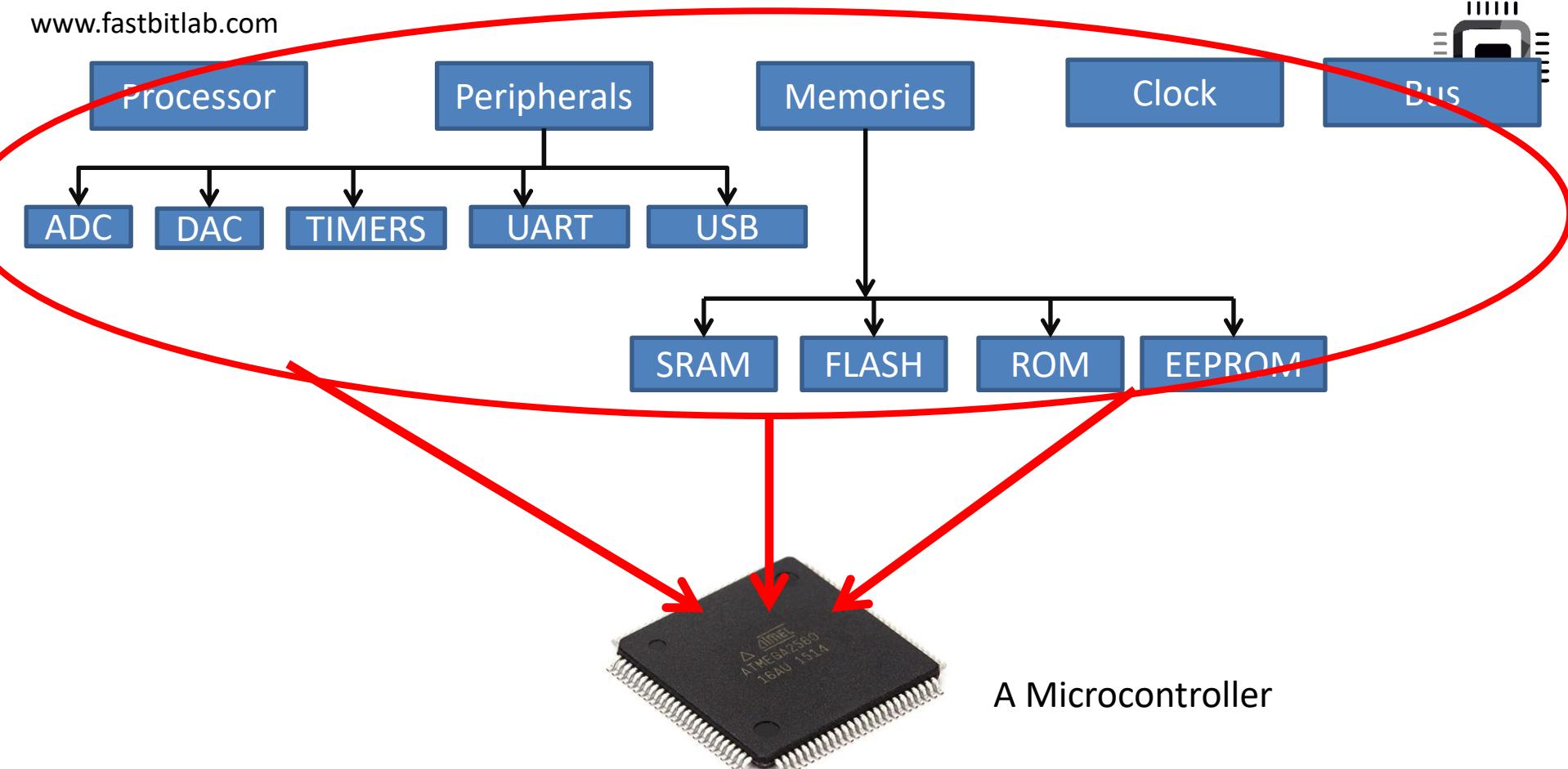
Analyzing Embedded C Program

- Anatomy of the microcontroller
- Identifying code and data parts of the program
- Code memory and data memory of the MCU
- Disassembly feature of the IDE
- Analyzing the executable(.elf) using GNU tools
(like objdump and size)

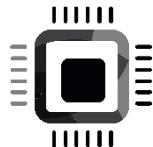


What is Microcontroller ?

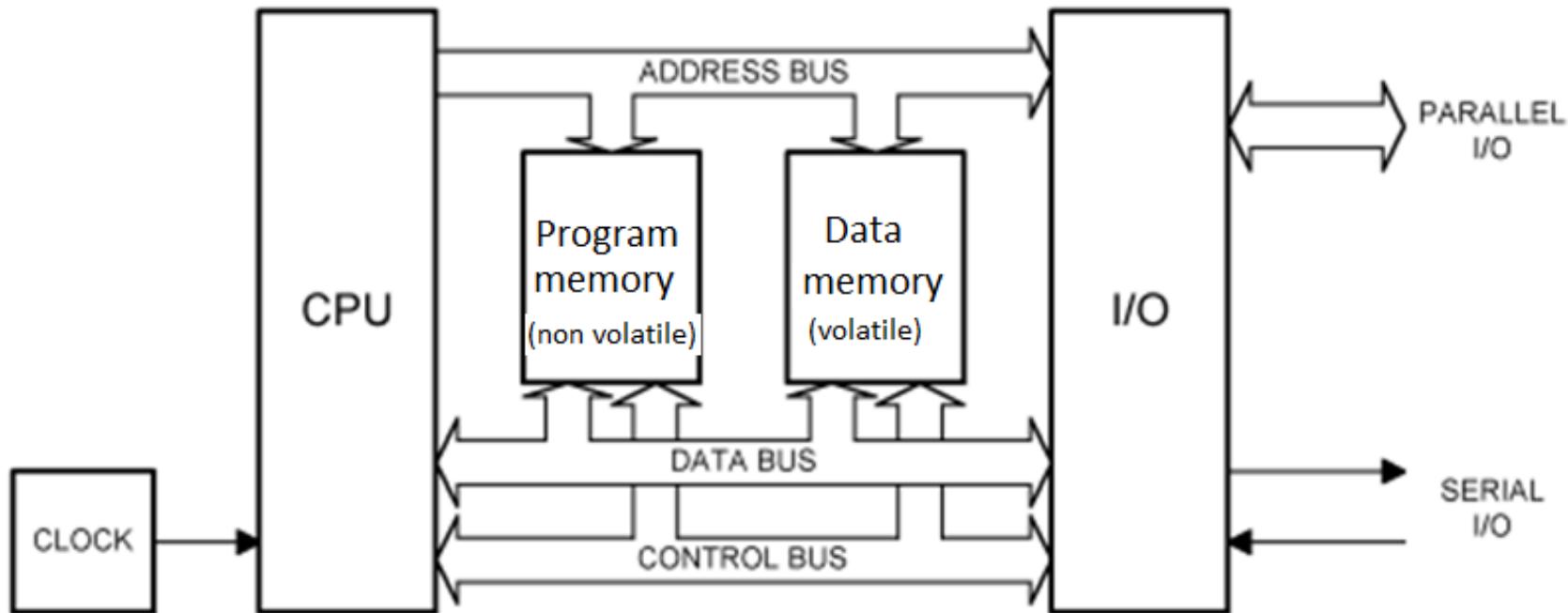
- Microcontroller (MCU, μ C) is a small computer system on a single chip. But its resources and capabilities such as memory, speed, external interfaces are very much limited than of a desktop computer because of MCU targets embedded applications.
- A typical microcontroller includes a processor, volatile and non-volatile memories, input/output(I/O) pins, peripherals, clock, bus interfaces on a single chip.

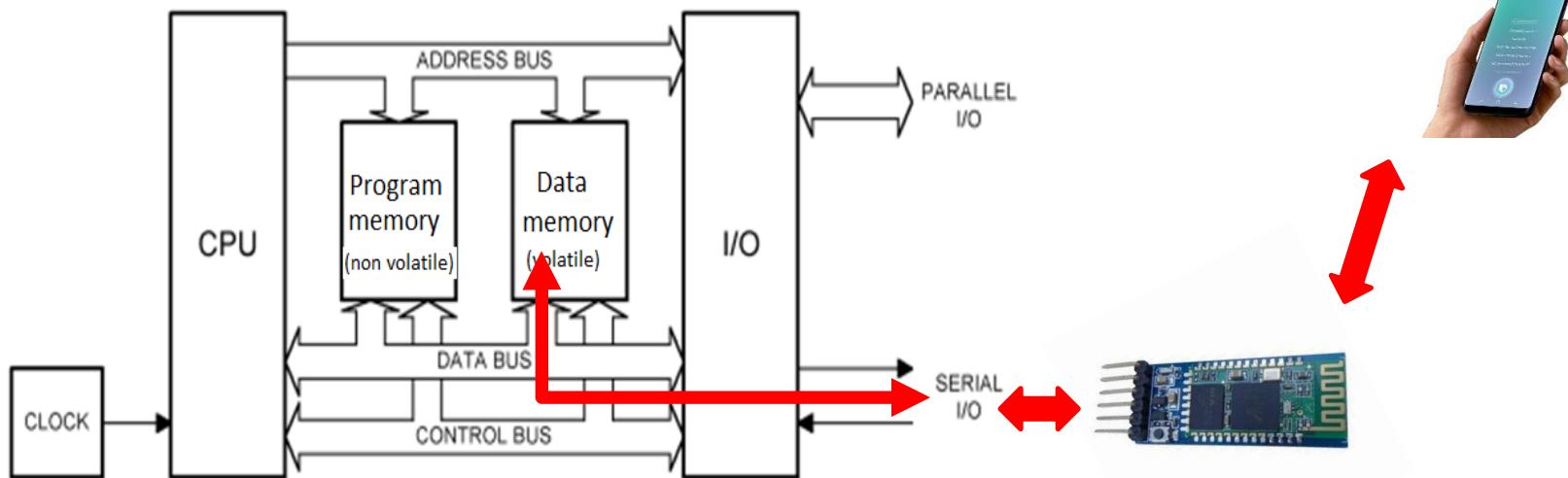
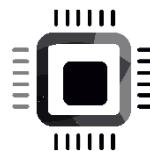


A Microcontroller

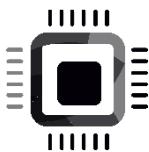


Anatomy of a Typical Small Microcontroller

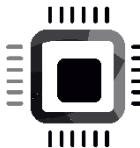




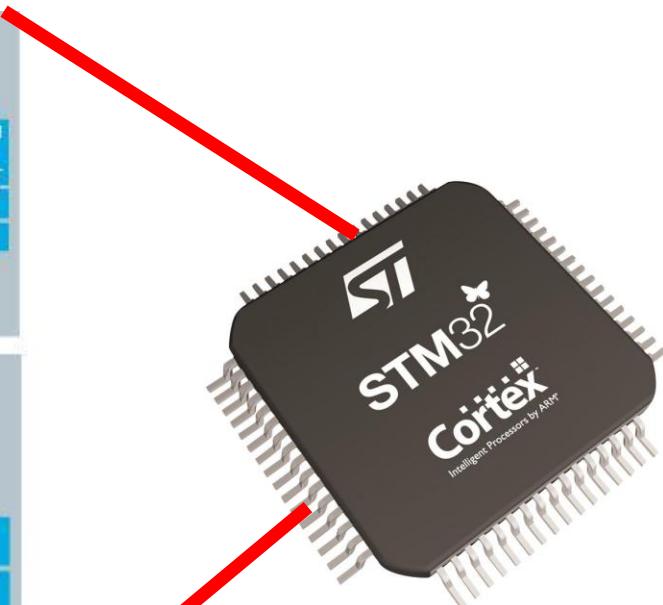
MCU receiving data from Bluetooth and storing in data memory

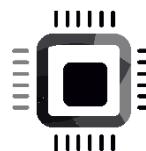


Inside view of the microcontroller

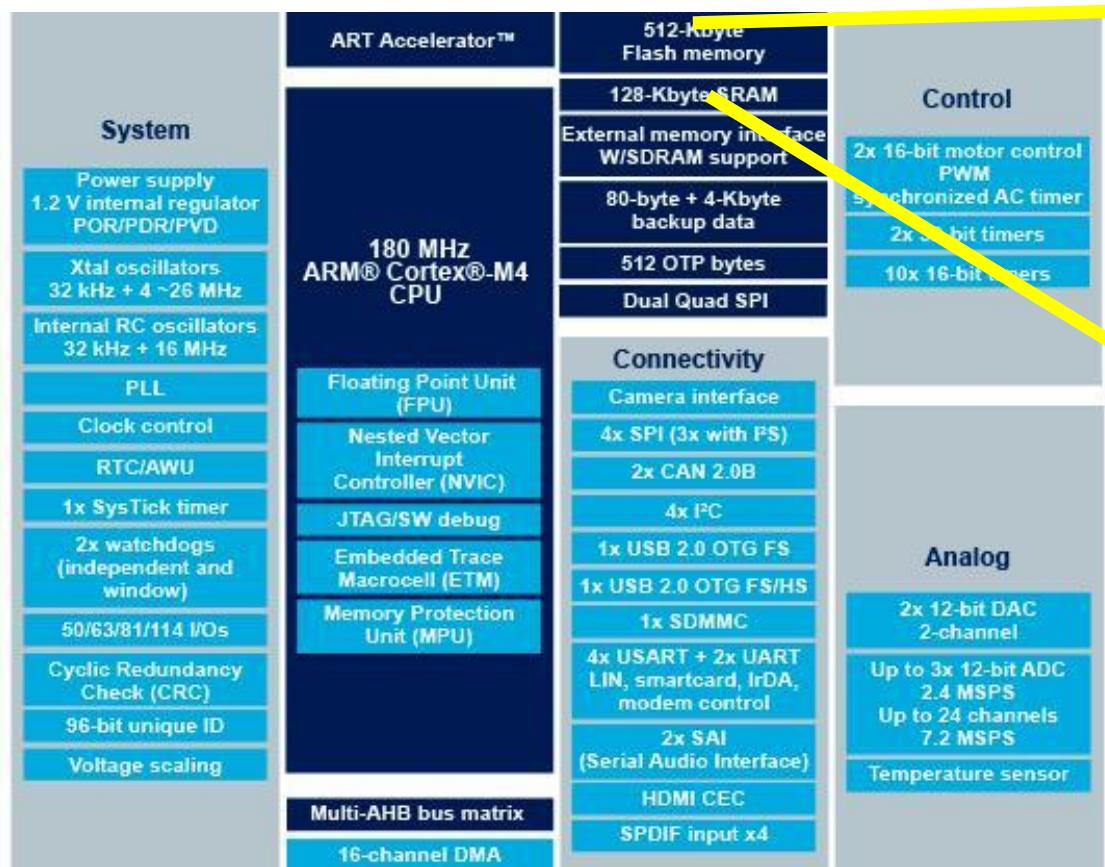
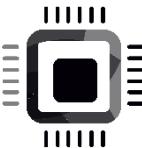


System	ART Accelerator™	Control
Power supply 1.2 V internal regulator POR/PDR/PVD	512-Kbyte Flash memory	
Xtal oscillators 32 kHz + 4 ~26 MHz	128-Kbyte SRAM	
Internal RC oscillators 32 kHz + 16 MHz	External memory interface W/SDRAM support	
PLL	80-byte + 4-Kbyte backup data	
Clock control	512 OTP bytes	
RTC/AWU	Dual Quad SPI	
1x SysTick timer	Connectivity	
2x watchdogs (independent and window)	Floating Point Unit (FPU)	
50/63/81/114 I/Os	Nested Vector Interrupt Controller (NVIC)	
Cyclic Redundancy Check (CRC)	JTAG/SW debug	
96-bit unique ID	Embedded Trace Macrocell (ETM)	
Voltage scaling	Memory Protection Unit (MPU)	
	Multi-AHB bus matrix	
	16-channel DMA	





System		ART Accelerator™	512-Kbyte Flash memory	Control
Power supply			128-Kbyte SRAM	
1.2 V internal regulator POR/PDR/PVD			External memory interface W/SDRAM support	
Xtal oscillators			80-byte + 4-Kbyte backup data	
32 kHz + 4 ~26 MHz			512 OTP bytes	
Internal RC oscillators			Dual Quad SPI	
32 kHz + 16 MHz				
PLL				
Clock control				
RTC/AWU				
1x SysTick timer		Floating Point Unit (FPU)	Camera interface	Analog
2x watchdogs (independent and window)		Nested Vector Interrupt Controller (NVIC)	4x SPI (3x with PS)	
50/63/81/114 IOs		JTAG/SW debug	2x CAN 2.0B	
Cyclic Redundancy Check (CRC)		Embedded Trace Macrocell (ETM)	4x I²C	
96-bit unique ID		Memory Protection Unit (MPU)	1x USB 2.0 OTG FS	
Voltage scaling			1x USB 2.0 OTG FS/HS	
			1x SDMMC	
			4x USART + 2x UART LIN, smartcard, IrDA, modem control	
			2x SAI (Serial Audio Interface)	
			HDMI CEC	
			SPDIF input x4	
		Multi-AHB bus matrix		
		16-channel DMA		



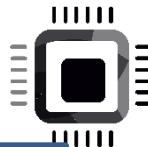
Code memory

Code Memory Type : Flash

data memory

Data Memory Type :
SRAM (Static RAM)

Inside view of Typical modern microcontroller



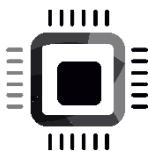
CPU
(ARM Cortex Mx)

Internal memories
(volatile and non
volatile)

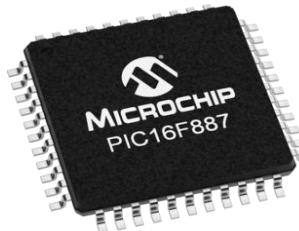
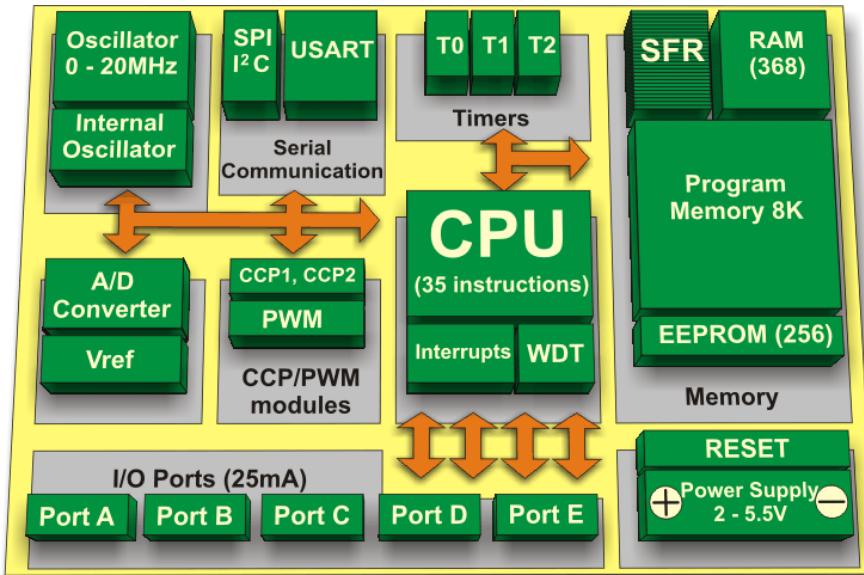
Voltage regulators
And other power
control ckts

Peripherals

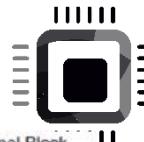
Clock producing
ckts



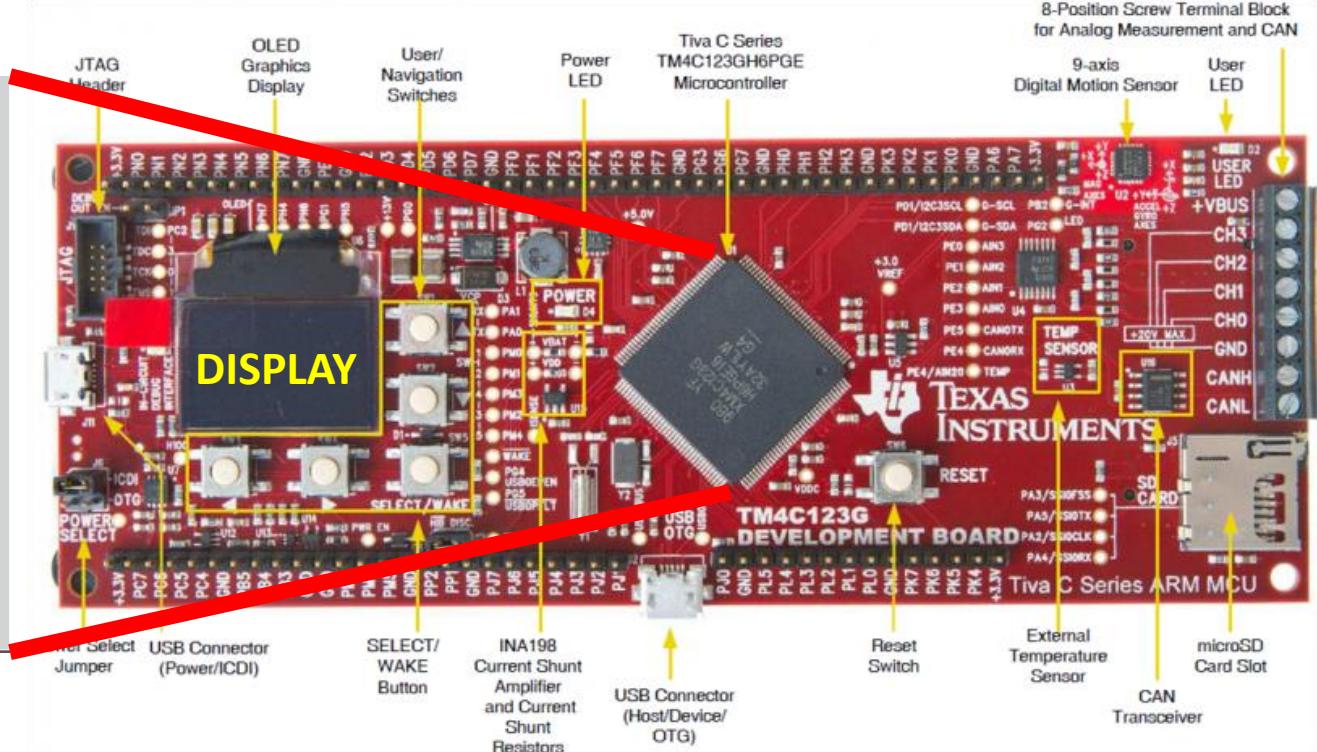
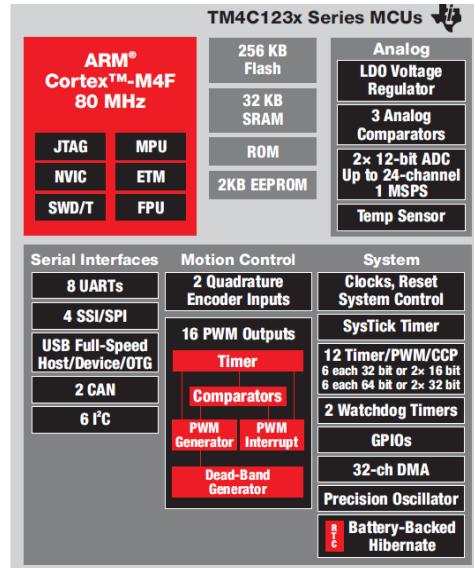
PIC16F887



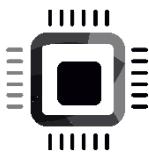
"Peripheral Interface Controller" made by Microchip Technology



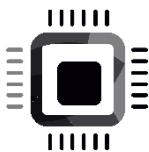
A Microcontroller



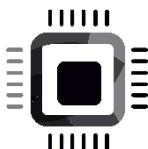
Microcontroller development board



Identifying code and data parts of the program

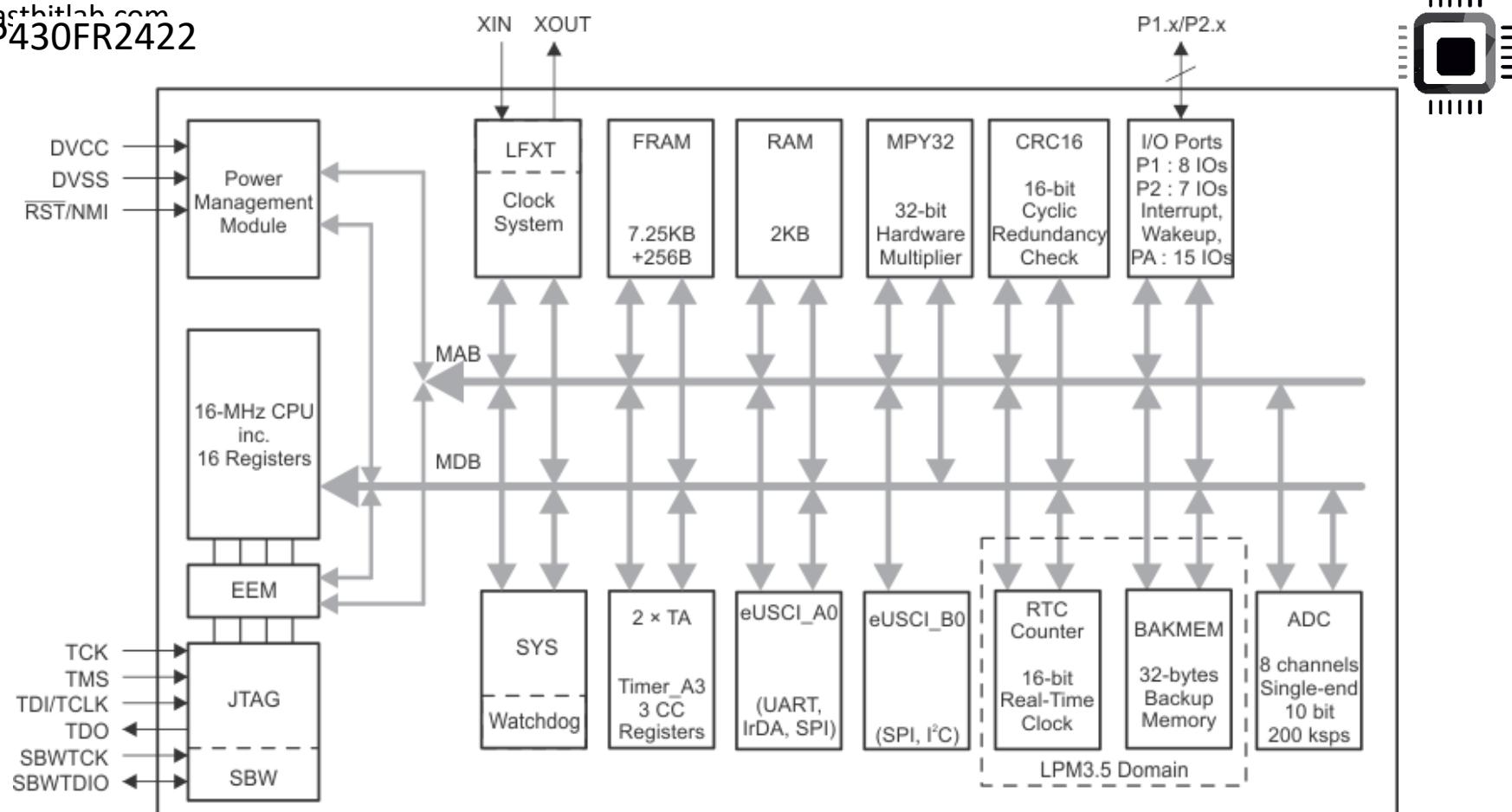


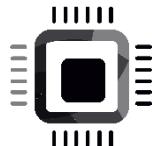
Code and data memory of the MCU



Code memory

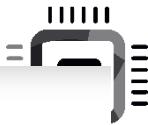
- The purpose of the code(Program) memory is to store instructions and constant data of your program.
- There are different types of code memory
 - ROM (Read only Memory)
 - MPROM (Mask Programmable Read only Memory)
 - EPROM (Ultraviolet Erasable Programmable ROM)
 - EEPROM (Electrically Erasable Programmable ROM)
 - OTP (On time programmable)
 - Flash
 - FRAM(Ferroelectric Random Access memory)





ROM

- Once code is stored inside the ROM, it can't be modified. Only reading is allowed.
- Suitable during MCU production with prewritten code targeting certain applications which will never be modified.
- Usage of ROM technology reduces cost. Because ROM is cheaper than Flash technology.



PIC16CR72 ☆

Status: Not Recommended for new designs

[View Datasheet](#)
[View Comparisons](#)

Features:

- ROM: 2048x14

CMOS ROM-based 8-bit microcontroller

Parametrics

Name	Value
Program Memory Type	ROM
Program Memory Size (KB)	3.5
CPU Speed (MIPS/DMIPS)	5
SRAM Bytes	128
Digital Communication Peri...	1-SPI, 1-I2C1-SSP(SPI/I2C)
Capture/Compare/PWM Per...	1 CCP,
Timers	2 x 8-bit, 1 x 16-bit
ADC Input	5 ch, 8-bit
Temperature Range (°C)	-40 to 85
Operating Voltage Range (V)	3 to 5.5
Pin Count	28



PIC18C452 ☆

Status: Not Recommended for new designs

[View Datasheet](#)
[View Comparisons](#)

Features:

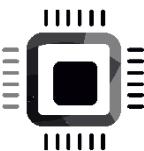
- Programmable Brown-Out, Programmable low voltage detect, Phase-Locked Loop (PLL), 8x8 Multiplier, 2 Capture/Compare, PSP

[View More](#)

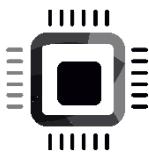
CMOS OTP-based 8-bit microcontroller

Parametrics

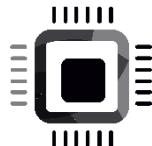
Name	Value
Program Memory Type	OTP
Program Memory Size (KB)	32
CPU Speed (MIPS/DMIPS)	10
SRAM Bytes	1,536
Digital Communication Peripherals	1-UART, 1-SPI, 1-I2C1-M
Capture/Compare/PWM Peripher...	2 Input Capture, 2 CCP,
Timers	1 x 8-bit, 3 x 16-bit
ADC Input	8 ch, 10-bit
Temperature Range (°C)	-40 to 125
Operating Voltage Range (V)	2.5 to 5.5
Pin Count	40



We downloaded the executable into program memory(FLASH) of the microcontroller. So, the data of our program must be in FLASH memory.
How come data arrived in data memory (SRAM) ?



Disassembly feature of the IDE



So , we can use objdump tool to disassemble the machine code generated

Dictionary

Search for a word



disassemble

/dɪsə'semb(ə)l/

verb

take (something) to pieces.

"the piston can be disassembled for transport"

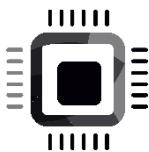
synonyms: [dismantle](#), take apart, take to pieces, pull apart, pull to pieces, take/pull to bits, [deconstruct](#), break up, strip down

the furniture was disassembled for transport

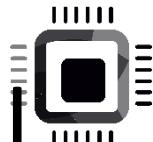
- COMPUTING

translate (a program) from machine code into a higher-level programming language.

"it is permissible for a lawful user to disassemble a computer program to determine its interfaces"



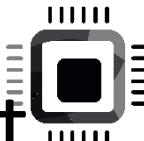
Analyzing the executable(.elf) using GNU tools



Analyzing .elf using objdump gnu tool

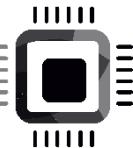
Run this command on your PC command prompt

arm-none-eabi-objdump.exe -h 003Add.elf



Processor Architecture and instruction set

- Processor : **ARM Cortex Mx (0,3,4,7....)**
- Processor architecture : **ARMv7-M**
- Instruction set architecture(ISA) : **Thumb-2 instruction set(16/32 bits instruction encoding)**

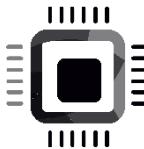


code

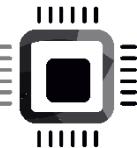
```
080001e8 <main>:  
080001e8:    b080      push   {r7, lr}  
080001ea:    b084      sub    sp, #16  
080001ec:    af00      add    r7, sp, #0  
080001ee:    2319      movs   r3, #23  
080001f0:    60fb      str    r3, [r7, #2]  
080001f2:    232d      movs   r3, #45 ; 0x2d  
080001f4:    60bb      str    r3, [r7, #8]  
080001f6:    4b0e      ldr    r3, [pc, #56] ; (8000230 <main+0x48>)  
080001f8:    607b      str    r3, [r7, #4]  
080001fa:    68fa      ldr    r2, [r7, #12]  
080001fc:    68bb      ldr    r3, [r7, #8]  
080001fe:    4413      add    r3, r2  
08000200:    461a      mov    r2, r3  
08000202:    687b      ldr    r3, [r7, #4]  
08000204:    601a      str    r2, [r3, #0]  
08000206:    687b      ldr    r3, [r7, #4]  
08000208:    681b      ldr    r3, [r3, #0]  
0800020a:    603b      str    r3, [r7, #0]  
0800020c:    f000 f82a  b1    8000264 <led_init>  
08000210:    f000 f84e  b1    80002b0 <turn_user_led_on>  
08000214:    4b07      ldr    r3, [pc, #28] ; (8000234 <main+0x4c>)  
08000216:    681b      ldr    r3, [r3, #0]  
08000218:    4618      mov    r0, r3  
0800021a:    f000 f80d  b1    8000238 <delay>  
0800021e:    f000 f857  b1    80002d0 <turn_user_led_off>  
08000222:    4b04      ldr    r3, [pc, #16] ; (8000234 <main+0x4c>)  
08000224:    681b      ldr    r3, [r3, #0]  
08000226:    4618      mov    r0, r3  
08000228:    f000 f806  b1    8000238 <delay>  
0800022c:    e7f0      b.n   8000210 <main+0x28>  
0800022e:    bf00      nop  
08000230:    20000044  .word 0x20000044  
08000234:    20000000  .word 0x20000000
```

So, our main function starts at address **0x080001e8** in the code memory

code



```
08000264 <led_init>:  
8000264: b480      push   {r7}  
8000266: af00      add    r7, sp, #0  
8000268: 4b0e      ldr    r3, [pc, #56] ; (80002a4 <led_init+0x40>)  
800026a: 681b      ldr    r3, [r3, #0]  
800026c: 4a0d      ldr    r2, [pc, #52] ; (80002a4 <led_init+0x40>)  
800026e: f043 0308  orr.w  r3, r3, #8  
8000272: 6013      str    r3, [r2, #0]  
8000274: 4b0c      ldr    r3, [pc, #48] ; (80002a8 <led_init+0x44>)  
8000276: 681b      ldr    r3, [r3, #0]  
8000278: 4a0b      ldr    r2, [pc, #44] ; (80002a8 <led_init+0x44>)  
800027a: f023 6340  bic.w  r3, r3, #201326592 ; 0xc000000  
800027e: 6013      str    r3, [r2, #0]  
8000280: 4b09      ldr    r3, [pc, #36] ; (80002a8 <led_init+0x44>)  
8000282: 681b      ldr    r3, [r3, #0]  
8000284: 4a08      ldr    r2, [pc, #32] ; (80002a8 <led_init+0x44>)  
8000286: f043 6380  orr.w  r3, r3, #67108864 ; 0x4000000  
800028a: 6013      str    r3, [r2, #0]  
800028c: 4b07      ldr    r3, [pc, #28] ; (80002ac <led_init+0x48>)  
800028e: 681b      ldr    r3, [r3, #0]  
8000290: 4a06      ldr    r2, [pc, #24] ; (80002ac <led_init+0x48>)  
8000292: f423 5300  bic.w  r3, r3, #8192 ; 0x2000  
8000296: 6013      str    r3, [r2, #0]  
8000298: bf00      nop  
800029a: 46bd      mov    sp, r7  
800029c: f85d 7b04  ldr.w  r7, [sp], #4  
80002a0: 4770      bx    lr  
80002a2: bf00      nop  
80002a4: 40023830  .word  0x40023830  
80002a8: 40020c00  .word  0x40020c00  
80002ac: 40020c04  .word  0x40020c04
```



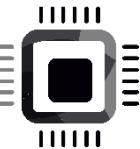
```
$ arm-none-eabi-objdump.exe -h LED_Project.elf

LED_Project.elf:      file format elf32-littlearm

Sections:
Idx Name      Size    VMA      LMA      File off  Align
 0 .isr_vector 000001ac 08000000 08000000 00010000 2**0
                CONTENTS, ALLOC, LOAD, READONLY, DATA
 1 .text       00000318 080001ac 080001ac 000101ac 2**2
                CONTENTS, ALLOC, LOAD, READONLY, CODE
 2 .rodata     00000004 080004c4 080004c4 000104c4 2**2
                CONTENTS, ALLOC, LOAD, READONLY, DATA
 3 .ARM.extab  00000000 080004c8 080004c8 00020434 2**0
                CONTENTS
 4 .ARM        00000000 080004c8 080004c8 00020434 2**0
                CONTENTS
 5 .preinit_array 00000000 080004c8 080004c8 00020434 2**0
                CONTENTS, ALLOC, LOAD, DATA
 6 .init_array  00000008 080004c8 080004c8 000104c8 2**2
                CONTENTS, ALLOC, LOAD, DATA
 7 .fini_array 00000004 080004d0 080004d0 000104d0 2**2
                CONTENTS, ALLOC, LOAD, DATA
 8 .data        00000434 20000000 080004d4 00020000 2**3
                CONTENTS, ALLOC, LOAD, DATA
 9 .bss         00000040 20000434 08000908 00020434 2**2
                ALLOC
10 ._user_heap_stack 00000404 20000474 08000908 00020474 2**0
                ALLOC
11 .ARM.attributes 00000030 00000000 00000000 00020434 2**0
                CONTENTS, READONLY
12 .debug_info   0000020c 00000000 00000000 00020464 2**0
                CONTENTS, READONLY, DEBUGGING
13 .debug_abbrev 0000011b 00000000 00000000 00020670 2**0
                CONTENTS, READONLY, DEBUGGING
14 .debug_aranges 00000080 00000000 00000000 00020790 2**3
                CONTENTS, READONLY, DEBUGGING
15 .debug_ranges 00000058 00000000 00000000 00020810 2**3
                CONTENTS, READONLY, DEBUGGING
16 .debug_macro  00001c86 00000000 00000000 00020868 2**0
                CONTENTS, READONLY, DEBUGGING
17 .debug_line   000006dc 00000000 00000000 000224ee 2**0
                CONTENTS, READONLY, DEBUGGING
```

Total code memory occupancy : 1224 bytes = 1.2KB

Total size of the instructions generated for our project : 792 bytes



data

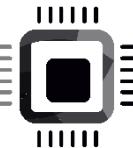
What is the data of our project ?

This time we are going to again analyze the elf file of our project

Because at the end of the day, this file should contain all the code and data right ?

Again, run the command

```
arm-none-eabi-objdump.exe -h LED_Project.elf
```



```
$ arm-none-eabi-objdump.exe -h LED_Project.elf

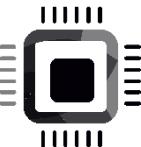
LED_Project.elf:      file format elf32-littlearm

Sections:
Idx Name      Size    VMA      LMA      File off  Align
 0 .isr_vector 000001ac 08000000 08000000 00010000 2**0
                CONTENTS, ALLOC, LOAD, READONLY, DATA
 1 .text       00000318 080001ac 080001ac 000101ac 2**2
                CONTENTS, ALLOC, LOAD, READONLY, CODE
 2 .rodata     00000004 080004c4 080004c4 000104c4 2**2
                CONTENTS, ALLOC, LOAD, READONLY, DATA
 3 .ARM.extab 00000000 080004c8 080004c8 00020434 2**0
                CONTENTS
 4 .ARM        00000000 080004c8 080004c8 00020434 2**0
                CONTENTS
 5 .preinit_array 00000000 080004c8 080004c8 00020434 2**0
                CONTENTS, ALLOC, LOAD, DATA
 6 .init_array 00000008 080004c8 080004c8 000104c8 2**2
                CONTENTS, ALLOC, LOAD, DATA
 7 .fini_array 00000004 080004d0 080004d0 000104d0 2**2
                CONTENTS, ALLOC, LOAD, DATA
 8 .data        00000434 20000000 080004d4 00020000 2**3
                CONTENTS, ALLOC, LOAD, DATA
 9 .bss         00000040 20000434 08000908 00020434 2**2
                ALLOC
10 ._user_heap_stack 00000404 20000474 08000908 00020474 2**0
                ALLOC
11 .ARM.attributes 00000030 00000000 00000000 00020434 2**0
                CONTENTS, READONLY
12 .debug_info   0000020c 00000000 00000000 00020464 2**0
                CONTENTS, READONLY, DEBUGGING
13 .debug_abbrev 0000011b 00000000 00000000 00020670 2**0
                CONTENTS, READONLY, DEBUGGING
14 .debug_aranges 00000080 00000000 00000000 00020790 2**3
                CONTENTS, READONLY, DEBUGGING
15 .debug_ranges 00000058 00000000 00000000 00020810 2**3
                CONTENTS, READONLY, DEBUGGING
16 .debug_macro 00001c86 00000000 00000000 00020868 2**0
                CONTENTS, READONLY, DEBUGGING
17 .debug_line   000006dc 00000000 00000000 000224ee 2**0
                CONTENTS, READONLY, DEBUGGING
```

we must be surprised
here to see , our project
already has 1076 bytes of
data

But can we locate those 1076
bytes of data in our project ?

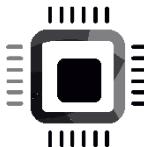
It's a kind of difficult task, lets
again use the command to analyze
further



data

```
$ arm-none-eabi-objdump.exe --syms LED_Project.elf | grep .data
080004c4 l      d  .rodata          00000000 .rodata
20000000 l      d  .data           00000000 .data
20000008 l      o  .data           00000428 impure_data
080004d4 g      *ABS* 00000000 _sidata
20000430 g      o  .data           00000004 __atexit_recursive_mutex
20000434 g      o  .data           00000004 __atexit_recursive_mutex
20000000 g      o  .data           00000004 g_DelayDuration
080004c4 g      o  .rodata          00000004 __global_impure_ptr
20000434 g      .data           00000000 _edata
```

These data will go to the data memory of your microcontroller



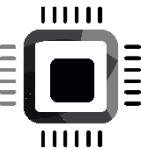
```
/*
 * This is function prototype.
 * prototype says that this function takes 2 input arguments of type int
 * and returns nothing (void)
 */
void function_add(int a , int b );

int main()
{
    //This is a function call . function is called three times
    function_add(90,10);
    function_add(100,10);
    function_add(-1,10);

}

/* This is function definition */
void function_add(int a , int b )
{
    int sum;
    sum = a+b;
    printf("Sum = %d\n",sum);

}
```



Argument passing

```
/*
 * This is function prototype.
 * prototype says that this function takes 2 input arguments of type int
 * and returns nothing (void)
 */
void function_add(int a , int b );

int main() Caller
{
    //This is a function call . function is called three times
    function_add(90,10);
    function_add(100,10);
    function_add(-1,10);

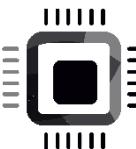
}

/* This is function definition */
void function_add(int a , int b )
{
    int sum;
    sum = a+b;
    printf("Sum = %d\n",sum);

}
```

Argument passing
(Call by value)

Callee



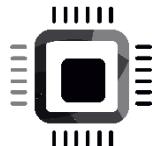
```
/*
 * This is function prototype.
 * prototype says that this function takes 2 input arguments of type int
 * and returns nothing (void)
 */
void function_add(int a , int b );

int main()
{
    //This is a function call . function is called three times
    function_add(90,10);
    function_add(100,10);
    function_add(-1,10);

}

/* This is function definition */
void function_add(int a , int b )
{
    int sum;
    sum = a+b;
    printf("Sum = %d\n",sum);
}
```

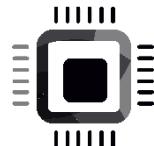
Local scope variables



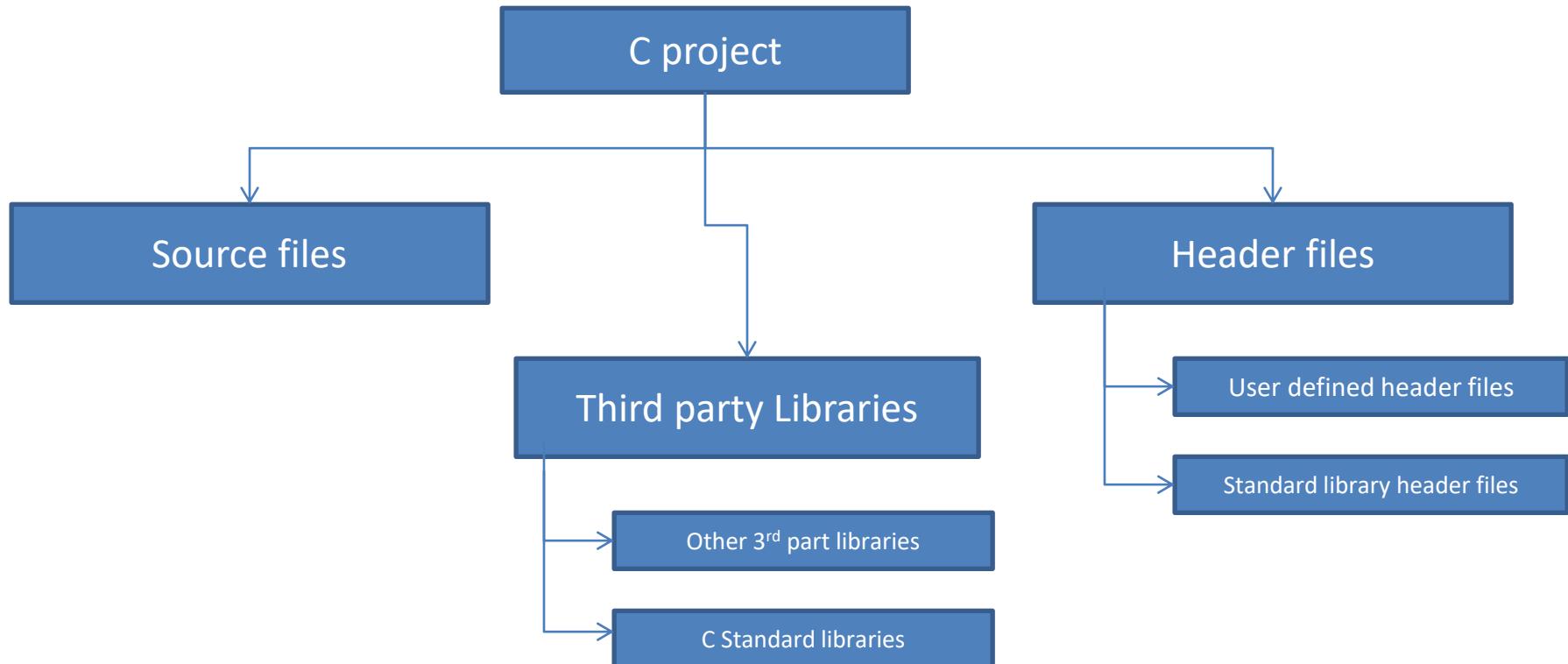
Typical C project structure

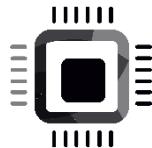
A C project is typically collection of 1 or more source files along with collection of optional header files.

Please note that header files are not compulsory for your project but most of the time header files bring more structure to our project and it is always recommended to use header files. Some times usage of header files become compulsory when you access C standard library codes or functions from your source file.

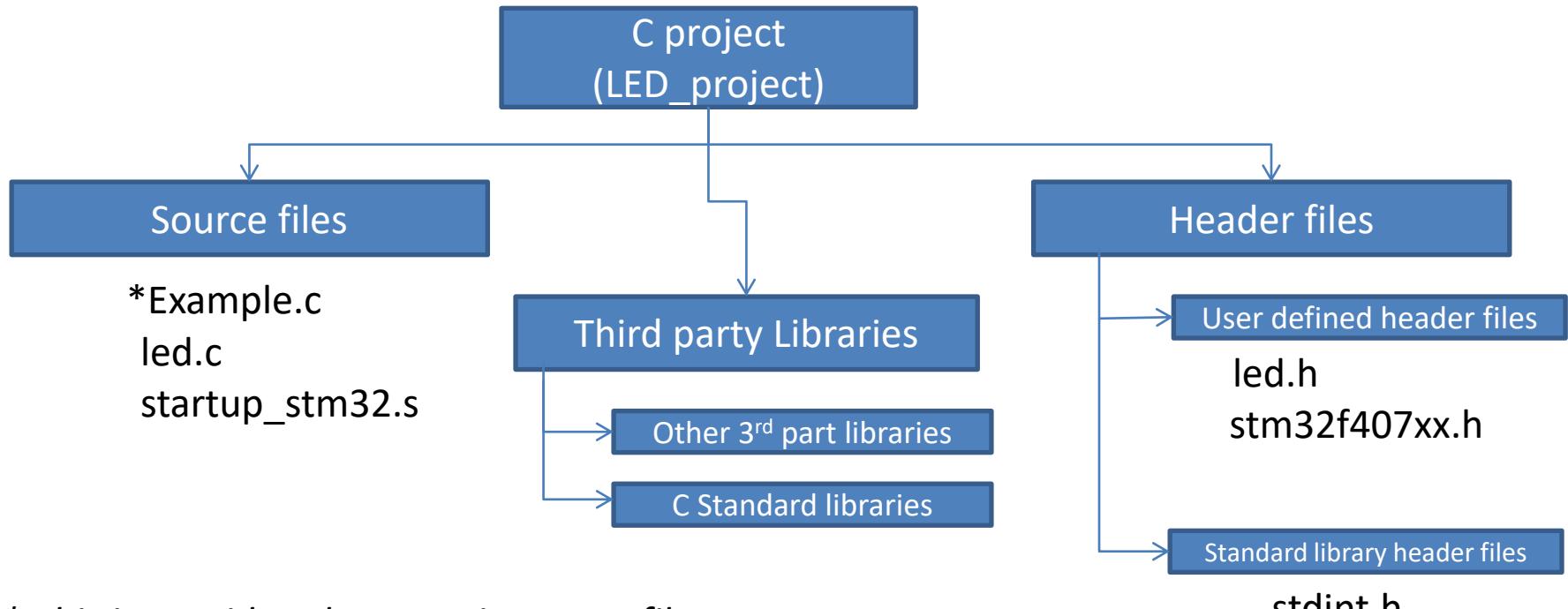


Typical C project structure

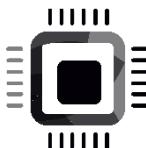




Typical C project structure



* This is considered as a main source file since it contains the main function of the project from where the execution starts



The main function

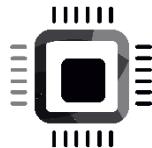
Every C program is a collection of various functions, implementing different functionalities.

But every C project will have at least one function whose name must be “main”
Main function is the starting point in your program execution *.

**Not really true , will discuss more on this later. But for a time being let's believe that main is starting point of execution of our c project*



The main function



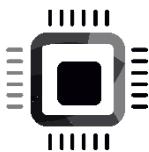
```
/* This is the starting point of the user application main.c*/
int main(void)
{
    /* These are the local variable declarations and initialization */
    int32_t a = 25;
    int32_t b = 45;

    /* This is a loop */
    while(1)
    { //Start of the loop body

        /* This is a function call */
        turn_user_led_on();
        /* This is a function call */
        delay(g_DelayDuration);
        /* This is a function call */
        turn_user_led_off();
        /* This is a function call */
        delay(g_DelayDuration);

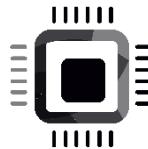
    } //end of the loop body

    //This is a return from the main function
    return 0;
}
```



```
|int main(void)
```

The main function



```
/* This is the starting point of the user application main.c*/
int main(void)
{
    /* These are the local variable declarations and initialization */
    int32_t a = 25;
    int32_t b = 45;

    /* This is a loop */
    while(1)
    { //Start of the loop body

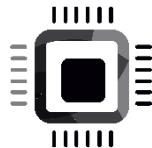
        /* This is a function call */
        turn_user_led_on();
        /* This is a function call */
        delay(g_DelayDuration);
        /* This is a function call */
        turn_user_led_off();
        /* This is a function call */
        delay(g_DelayDuration);

    } //end of the loop body

    //This is a return from the main function
    return 0;
}
```

Function body

The main function



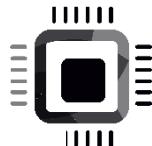
```
/* This is the starting point of the user application main.c*/
int main(void)
{
    /* These are the local variable declarations and initialization */
    int32_t a = 25;
    int32_t b = 45;

    /* This is a loop */
    while(1)
    { //Start of the loop body

        /* This is a function call */
        turn_user_led_on();
        /* This is a function call */
        delay(g_DelayDuration);
        /* This is a function call */
        turn_user_led_off();
        /* This is a function call */
        delay(g_DelayDuration);

    } //end of the loop body

    //This is a return from the main function
    return 0;
}
```



From c99 standard about main

5.1.2.2.1 Program startup

- 1 The function called at program startup is named **main**. The implementation declares no prototype for this function. It shall be defined with a return type of **int** and with no parameters:

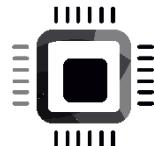
```
int main(void) { /* ... */ }
```

or with two parameters (referred to here as **argc** and **argv**, though any names may be used, as they are local to the function in which they are declared):

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

or equivalent;⁹⁾ or in some other implementation-defined manner.

- 2 If they are declared, the parameters to the **main** function shall obey the following constraints:
 - The value of **argc** shall be nonnegative.



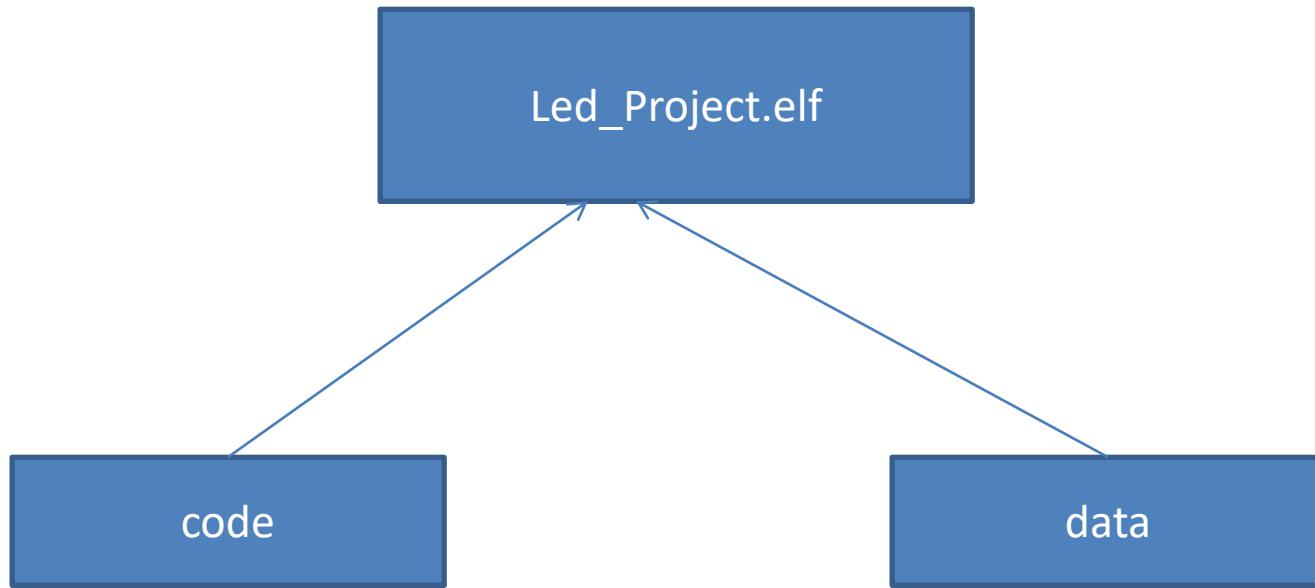
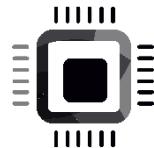
Executables

Led_project.elf

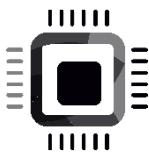
Led_project.bin

Lets analyze the elf executable generated to understand what are the codes and data hidden in our project .

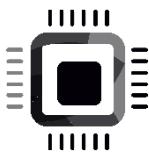
For that we are going to use the tool called objdump



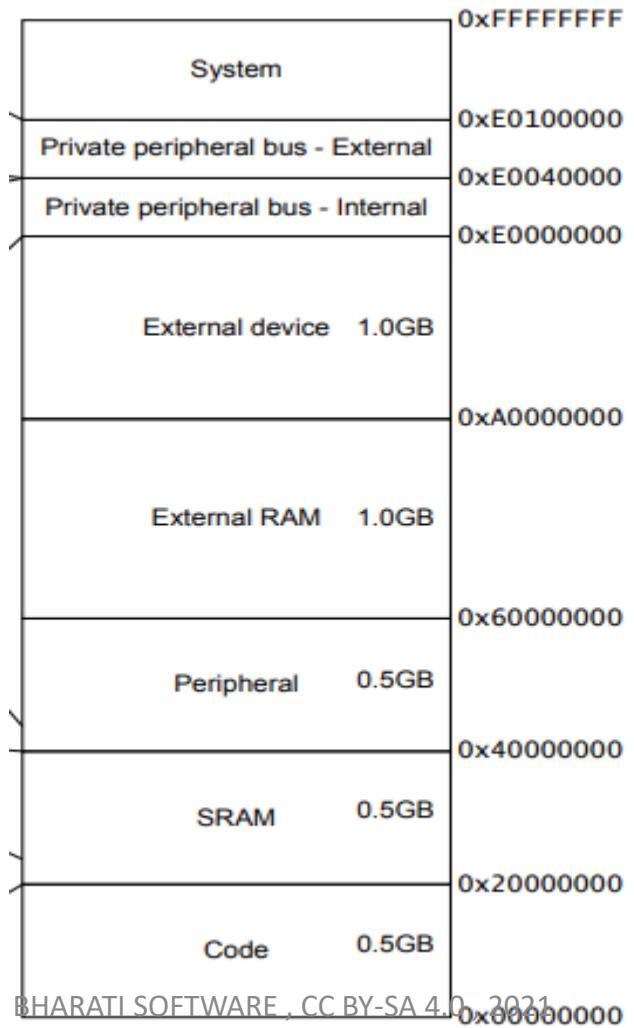
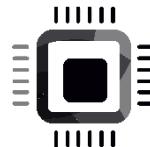
Now, fortunately there is one command which we can run to analyze the elf files . To understand code section and data section

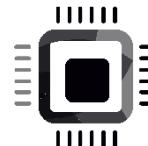


Memory map

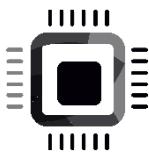


Processor core
Register sets



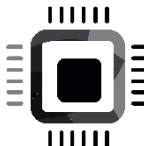


Memory Map	Region
Code	Instruction fetches are performed over the ICode bus. Data accesses are performed over the DCode bus.
SRAM	Instruction fetches and data accesses are performed over the system bus.
SRAM bit-band	Alias region. Data accesses are aliases. Instruction accesses are not aliases.
Peripheral	Instruction fetches and data accesses are performed over the system bus.
Peripheral bit-band	Alias region. Data accesses are aliases. Instruction accesses are not aliases.
External RAM	Instruction fetches and data accesses are performed over the system bus.
External Device	Instruction fetches and data accesses are performed over the system bus.
Private Peripheral Bus	External and internal <i>Private Peripheral Bus</i> (PPB) interfaces. This memory region is <i>Execute Never</i> (XN), and so instruction fetches are prohibited. An MPU, if present, cannot change this.
System	System segment for vendor system peripherals. This memory region is XN, and so instruction fetches are prohibited. An MPU, if present, cannot change this.

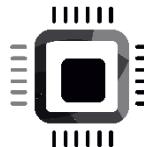


Manipulating decimal numbers in ‘C’

Representation of decimal numbers in ‘C’



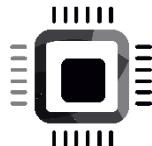
- A Decimal Number contains a Decimal Point.
 - 125.55 is a decimal number (a real number)
- In computer memory, the real numbers are stored according to the representation standardized by the IEEE standard 754
- IEEE754 floating-point representation is an approximate representation of real numbers.
- All computer systems and microcontrollers nowadays use this standard to store real numbers in memory
- If you are working with numbers that have a fractional part or in case you are using integers that don't fit into a long data type, then we can use floating-point representation.



Real numbers

- Too small real number
 - Charge of an electron
 - $-1.60217662 \times 10^{-19}$ coulombs
- Too big real number
 - distance between earth and andromeda galaxy
 - 2.3651826×10^{19} Kms

In 'C' too small or too big numbers or numbers having a fractional part are represented using floating-point representation data types such as float and double

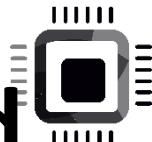


Why do we need floating point representation?

- To store, a too-small number in memory
- To store, a too big number in memory

Think of three situations to store decimal number

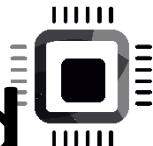
- 1) Storing charge of an electron (too small)
- 2) Storing light years to kilometers conversion
- 3) Which data type do you use to store 1.767×10^{100} (too big)?



The IEEE-754 floating-point standard

- The IEEE-754 is a standard for representing and manipulating floating-point quantities that are followed by all modern computer systems and microcontrollers.

$+7.432 \times 10^{48}$

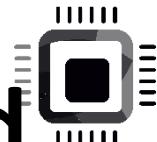


The IEEE-754 floating-point standard

+7.432 x 10⁴⁸

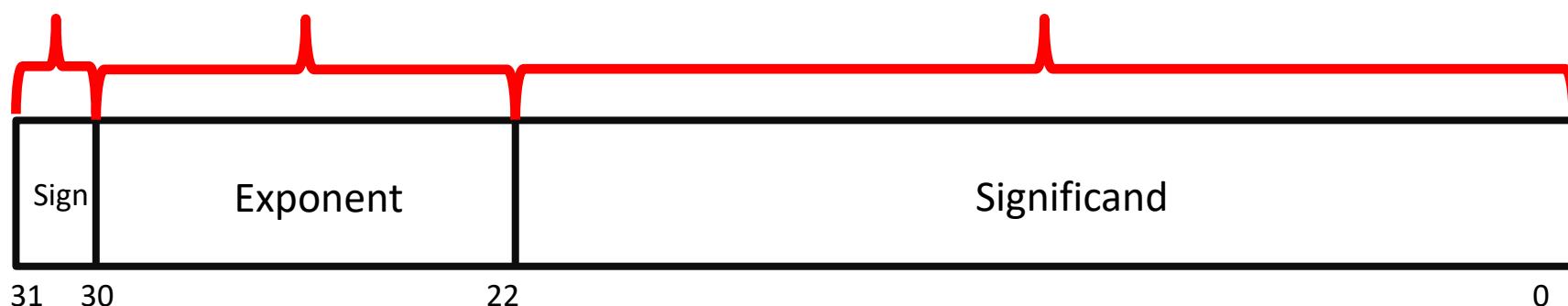
The diagram illustrates the IEEE-754 floating-point format. It shows the number $+7.432 \times 10^{48}$. A red oval encloses the significand 7.432 , which is labeled "Mantissa (Significand)". Another red oval encloses the exponent 48 , which is labeled "Exponent". Arrows point from the labels "sign", "Mantissa (Significand)", and "Exponent" to their respective components in the number.

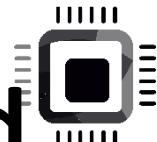
sign Mantissa (Significand) Exponent



The IEEE-754 floating-point standard

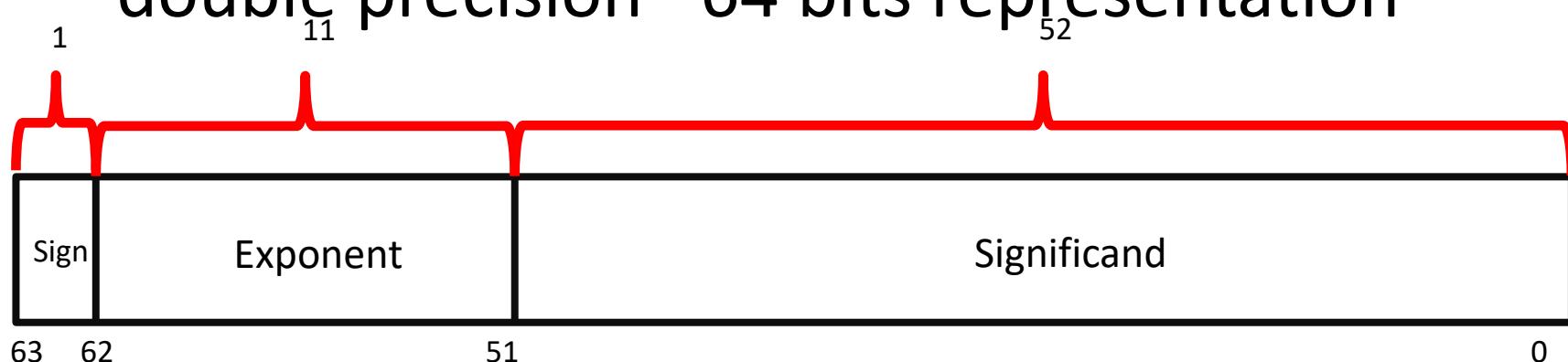
- Single precision → 32 bits representation

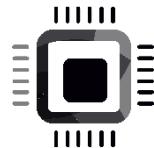




The IEEE-754 floating-point standard

- double precision – 64 bits representation

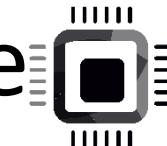




Integer part Decimal point fractional part

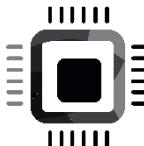
125.55

- You cannot represent this number in a program using integer data types like int, char, long.
- You need particular data types to represent these decimal numbers
 - Float (32-bit floating-point representation, single precision)
 - Double (64-bit floating-point representation, double precision)



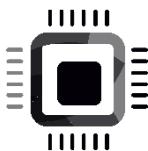
Format specifier for float and double data types

- Use **%lf** format specifier to read or write **double** type variable
- Use **%f** format specifier to read or write **float** type variable
- Use **%e** **%l** format specifier to read or write real numbers in scientific notation
- All constants with a decimal point are considered as **double** by default



Hardware floating point unit

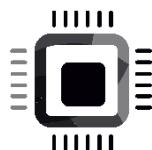
- ARM Cortex processor provides inbuilt optional floating-point units that can manipulate floating-point data which boosts the performance.
- Without a hardware floating unit, you need to use math libraries to manipulate floating-point numbers stored in the memory which consumes way too many clock cycles and reduces the performance for embedded systems.
- So, if your application deals with too frequent access to floating-point numbers then selecting a microcontroller that supports the hardware floating-point unit may drastically improve the performance of your application in terms of speed, memory and power consumption.



Exercise

- Find out the number of electrons responsible for producing the given charge. Use scientific notation while inputting and outputting the numbers.

Number of electrons = given charge / charge of Electron

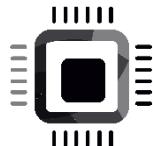


Range of float

Storage size : 4 bytes

Precision : up to 6 decimal places

Value range : 1.2×10^{-38} to 3.4×10^{38}

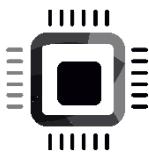


Range of double

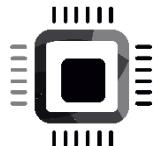
Storage size : 8 bytes

Precision : up to 15 decimal places

Value range : 2.3×10^{-308} to 1.7×10^{308}



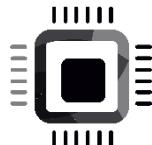
Type Qualifiers in ‘C’



Type Qualifiers in ‘C’

1. const
2. volatile

Applying these qualifiers to a variable declaration is called *qualifying* the declaration

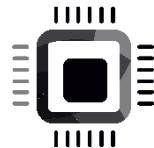


‘const’ type qualifier

- ‘const’ is a type qualifier in ‘C’ used to enforce read-only feature on variables.

```
uint8_t data1 = 10; //here 'data1' is called a variable  
data1 = 50;           // OK. 'data1' value can be modified through out the program
```

```
uint8_t const data2 = 10; //here 'data2' is called constant variable  
data2 = 50;    //Compile-time- error ! 'data2' value can not be modified because it is read-only
```

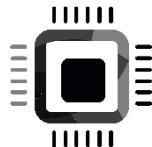


Type specifier of a variable

Read-only(constant) variable

```
uint8_t const data1 = 10;
```

Type qualifier
of a variable



```
const uint8_t data1 = 10; }  
uint8_t const data1 = 10; }
```

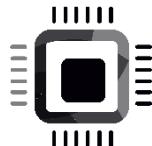
Both statements are identical

```
uint8_t const data1 = 10; }
```

This is preferred



"data1" is a read-only(constant) variable of type unsigned integer_8



About “const”ness of a variable

- By using the ‘const’ keyword, you are just making a promise to the compiler that you(the programmer) won't try to modify the content of the variable using its name
- If you try to modify the variable by its name, the compiler stops you by throwing an error. (**compile-time error**)
- You can still modify the content of the variable by using its address



'const' doesn't mean that the value never changes, its only programming safety feature to ensure that the programmer shouldn't try to modify the value.

```
int main(void)
{
    uint8_t const data = 10;

    data = 50; //This is Error

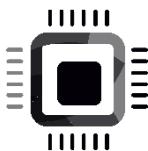
    printf("Value of data = %u\n",data);
}
```

```
int main(void)
{
    uint8_t const data = 10;

    uint8_t *ptr = (uint8_t*)&data;

    *ptr = 50;

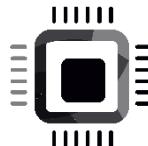
    printf("Value of data = %u\n",data);
}
```

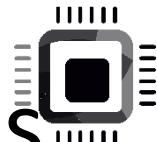


'const' doesn't mean that the value never changes, its only programming safety feature to ensure that the programmer shouldn't try to modify the value.

Placement of 'const' variables in memory

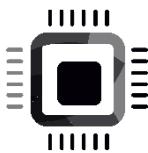
- All local const variables are just like non-const variables as far as memory placement is concerned. They are placed in RAM. The only specialty of a const variable is, it is read-only
- All Global const variables are stored in ROM or FLASH. This also further depends on linker script rules and the hardware on which code runs.
- In STM32 target hardware, all global const variables live in FLASH memory. So, when you try to modify the const variable using its address, operation has no effect. Because flash memory of the microcontroller is write-protected.



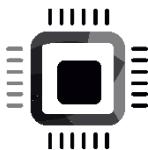


Local ‘const’ vs global ‘const’ variables

Local ‘const’ variable	Global ‘const’ variable
Stored in RAM with local scope	stored in ROM or FLASH
Can be modified using variable address	Modifying variable value using its address has undefined behavior or no effect



Const pointer and different case study



Case1: constant data

```
uint8_t const data = 50;
```

//This is a case of const data

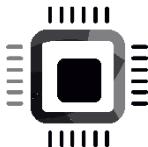
Use cases:

To define mathematical constants in the program

```
float const pi = 3.1415;
```

```
float const radius = 4;
```

```
int const number_of_months = 12;
```



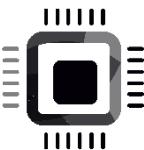
Case2: Modifiable pointer and constant data

```
uint8_t const *pData = (uint8_t*) 0x40000000;
```

Read like this

- Here the pointer pData is modifiable but the data pointed by the pData cannot be modifiable (read-only).
- So, we can say that pData is a pointer pointing to read-only data

pData is a pointer (*) pointing to constant data of type unsigned integer_8



uint8_t const *pData = (uint8_t*) 0x40000000;

Allowed

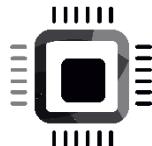
`pData = (uint8_t*) 0x50000000;`

Not-allowed

`*pData = 50;`

`pData = (uint8_t*) 0x60000000;`

`pData = (uint8_t*) 0x70000000;`



Case2: Modifiable pointer and constant data

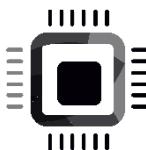
```
uint8_t const *pData = (uint8_t*) 0x40000000;
```

```
//This function copies data from src pointer to dst pointer
void copy_src_to_dst(uint8_t *src, uint8_t *dst, uint32_t len)
{
    for(uint32_t i = 0 ; i < len ; i++)
    {
        *dst = *src; //*dst++ = *src++
        dst++;
        src++;
    }
}
```

src is not guarded (prone to mistake)

```
//This function copies data from src pointer to dst pointer
void copy_src_to_dst(uint8_t const *src, uint8_t *dst, uint32_t len)
{
    for(uint32_t i = 0 ; i < len ; i++)
    {
        *dst = *src; //*dst++ = *src++
        dst++;
        src++;
    }
}
```

src is guarded (compiler alerts if programmer tries to change the data pointed by src pointer)



open(3) - Linux man page

Prolog

This manual page is part of the POSIX Programmer's Manual. The Linux implementation of this interface may differ (consult the corresponding Linux manual page for details of Linux behavior), or the interface may not be implemented on Linux.

Name

open - open a file

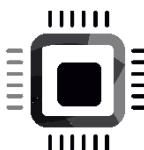
Synopsis

```
#include <sys/stat.h>
#include <fcntl.h>

int open(const char *path, int oflag, ... );
```

Description

The *open()* function shall establish the connection between a file and a file descriptor.



write(2) - Linux man page

Name

write - write to a file descriptor

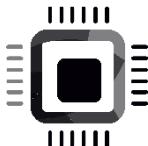
Synopsis

```
#include <unistd.h>
```

```
ssize_t write(int fd, const void *buf, size_t count);
```

Description

`write()` writes up to *count* bytes from the buffer pointed *buf* to the file referred to by the file descriptor *fd*.



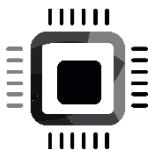
Case3: Modifiable data and constant pointer

```
uint8_t *const pData = (uint8_t*) 0x40000000;
```

Read like this

- Here the pointer pData is read-only but the data pointed by the pData can be modifiable.
- So, we can say that pData is a read only pointer pointing to modifiable data

pData is a constant pointer (*) pointing to data of type unsigned integer_8



uint8_t * const pData = (uint8_t*) 0x40000000;

Not - Allowed

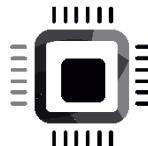
pData = (uint8_t*) 0x50000000;

Allowed

***pData = 50;**

***pData = 10;**

***pData = 60;**



Case3: Modifiable data and constant pointer

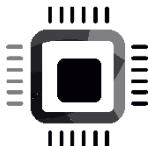
```
uint8_t *const pData = (uint8_t*) 0x40000000;
```

```
/*
 *Update the details of age and salary in to the pointer provided
 *by the caller
 */
void update_user_data(uint8_t *const pUserAge , uint32_t *const pUserSalary)
{
    if(pUserAge != NULL){
        *pUserAge = getUserAge();
    }

    if(pUserSalary != NULL){
        *pUserSalary = getUserSalary();
    }
}
```

Use case :

Improve the readability
and guard the pointer
variables



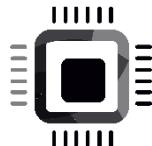
Case4: const data and const pointer

```
uint8_t const *const pData = (uint8_t*) 0x40000000;
```

Read like this

- Here the pointer pData is read-only and the data pointed by the pData is also read-only
- So, we can say that pData is a read only pointer pointing to read-only data

pData is a constant pointer (*) pointing to constant data of type unsigned integer_8

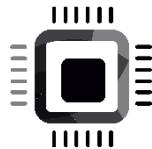


Case4: const data and constant pointer

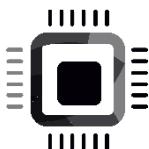
```
uint8_t const *const pData = (uint8_t*) 0x40000000;
```

```
/*
 * read and return the content of status register pointed by
 * pStatusReg
 * accidental write to SR may cause unpredictable consequences|
 */
uint32_t read_status_register(uint32_t const *const pStatusReg)
{
    return (*pStatusReg);
}
```

Uses of ‘const’

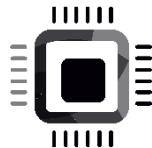


- It adds some safety while you write code. The compiler warns you when trying to change the value of the const variable.
- Since a constant variable doesn't change, It has only one state throughout the problem. So, you need not track its various states.
- Improves the readability of your program.
- Use it generously to enforce pointer access restrictions while writing functions and function prototypes
- It may also help compiler to generate optimized code.



Type Qualifiers in ‘C’

1. const
2. volatile

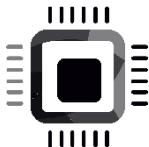


Volatile

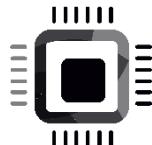
Volatile is a type qualifier in ‘C’ used with variables to instruct the compiler not to invoke any optimization on the variable operation.

It tells the compiler that the value of the variable may change at any time with or without the programmer’s consent. So, the compiler turns off optimizing the read-write operations on variables which are declared using volatile keyword

Volatile is very much helpful in embedded systems codes

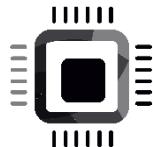


- Example of volatile effect
- Data1 = data2;
- Data1 = data2;



When to use ‘Volatile’ qualifier ?

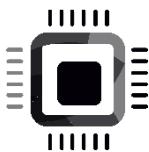
- A variable must be declared using a volatile qualifier when there is a possibility of unexpected changes in the variable value.
- The unexpected changes in the variable value may happen from within the code or from outside the code (from the hardware)



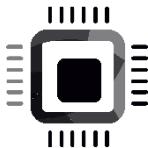
When to use ‘Volatile’ qualifier ?

Use volatile when your code is dealing with below scenarios

1. **Memory-mapped peripheral registers** of the microcontrollers
2. Multiple tasks accessing global variables(read/write) in an RTOS multithreaded application
3. When a global variable is used to share data between the main code and an ISR code.



Syntax of using 'volatile'

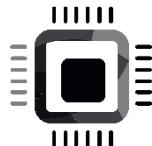


Case1: volatile data

```
uint8_t volatile my_data;  
volatile uint8_t my_data;
```

Read like
this

my_data is a volatile variable of type unsigned interger_8



Case2: non-volatile pointer to volatile data

Read like
this

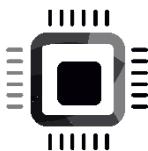
```
uint8_t volatile *pStatusReg;
```

pStatusReg is a non-volatile pointer , pointing to volatile data of type unsigned interger_8

Use case :

This is a perfect case of accessing memory-mapped registers.

Use this syntax generously whenever you are accessing memory mapped registers in your microcontroller code



Case3: volatile pointer to non-volatile data

```
uint8_t *volatile pStatusReg;
```

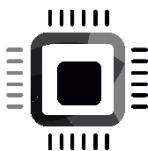
pStatusReg is a volatile pointer , pointing to non-volatile data of type unsigned interger_8

Rarely used

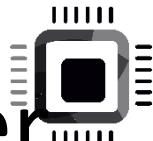
Case4: volatile pointer to volatile data

```
uint8_t volatile *volatile pStatusReg;
```

pStatusReg is a volatile pointer , pointing to volatile data of type unsigned interger_8



The keywords 'const' and 'volatile' can be applied to any declaration, including those of structures, unions, enumerated types or typedef names.

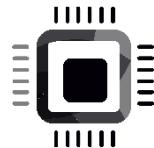


Usage of ‘const’ and ‘volatile’ together

- You can also use both ‘const’ and ‘volatile’ qualifiers in a variable declaration as per your goal.
- Example

```
uint8_t volatile *const pReg = (uint8_t*)0x40000000;
```

```
uint8_t const volatile *const pReg = (uint8_t*)  
0x40000000
```



Case of reading from read-only buffer or address which is prone to unexpected change

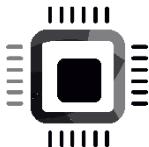
`uint8_t const volatile *const pReg = (uint8_t*) 0x40000000`

Data coming from external world

0xF1

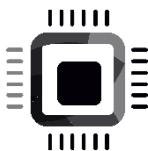
Input data register of a peripheral

Or a shared memory from which you are supposed to read-only



Structures in 'C'

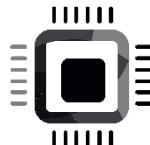
- Structure is a data structure used to create user-defined data types in 'c'
- Structures allow us to combine data of different types



Creation of structure

```
struct tag_name  
{  
    member_element-1;  
    member_element-2;  
    member_element-3;  
    member_element-n;  
};
```

This is syntax
to create a
structure in C



Example of a structure

```
struct CarModel
```

```
{
```

```
    unsigned int
```

```
    carNumber;
```

```
    uint32_t
```

```
    carPrice;
```

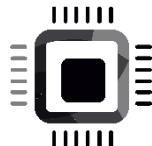
```
    uint16_t
```

```
    carMaxSpeed;
```

A large red curly brace is positioned to the right of the code, spanning from the opening brace '{' to the final closing brace '}' at the end of the structure definition.

This called structure definition

Structure definition doesn't consume any memory
.
Its just a description or a record



Variables of a structure

struct CarModel

{

```
unsigned int carNumber;  
uint32_t carPrice;  
uint16_t carMaxSpeed;  
float carWeight;
```

};

User defined data type

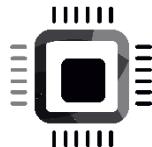
Structure variables

struct CarModel CarBMW, CarFord , CarHonda ;

*(Memory will be consumed when
you create structure variables.)*

What is the data type of CarBMW ?

The data type of CarBMW variable is “**struct CarModel**”



Structure member element initialization

struct CarModel

{

 unsigned int carNumber;

 uint32_t carPrice;

 uint16_t carMaxSpeed;

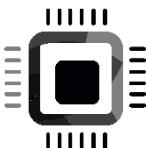
 float carWeight;

struct CarModel CarBMW ={2021,15000,220,1330 };// C89 method . Order is important

};

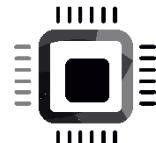
struct CarModel CarBMW ={.carNumber= 2021,.carWeight =1330 ,.carMaxSpeed =220,.carPrice =15000 };

//C99 method using designated initializers



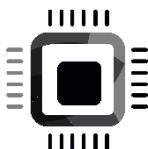
Example

- Write a program to create a **carModel** structure discussed and create 2 variables of type **carModel**. Initialize the variables with the below given data and then print them
 1. 2021,15000,220,1330
 2. 4031,35000,160,1900.96



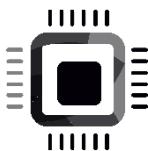
Accessing Structure Members

When a structure variable is created , use a **.(dot)** operator to access the member elements

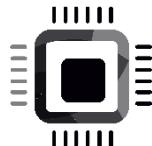


Example

- Write a program to create a **carModel** structure discussed and create 2 variables of type **carModel**. Initialize the variables with the user given data and then print all member elements .

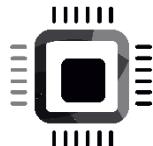


Aligned/un-aligned data access and structure padding



Aligned/un-aligned data access

- For efficiency, the compiler generates instructions to store variables on their natural size boundary addresses in the memory
- This is also true for structures. Member elements of a structure are located on their natural size boundary



Natural size boundary

Char

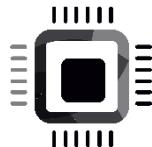
Address	0403010	0403011	0403012	0403013	0403014	0403015
---------	---------	---------	---------	---------	---------	---------

short

Address	0403010	0403012	0403014	0403016	0403018	040301A
---------	---------	---------	---------	---------	---------	---------

int

Address	0403010	0403014	0403018	040301C	0403020	0403024
---------	---------	---------	---------	---------	---------	---------



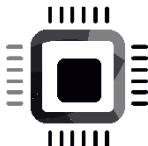
Natural size boundary

char a char b char c char d

Address	0403010	0403011	0403012	0403013	0403014	0403015	0403016	0403017	0403018
---------	---------	---------	---------	---------	---------	---------	---------	---------	---------

char a short b short c

Address	0403010	0403011	0403012	0403013	0403014	0403015	0403016	0403017	0403018
---------	---------	---------	---------	---------	---------	---------	---------	---------	---------

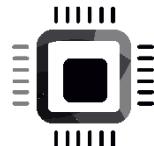


Aligned data

```
//Some global variables to understand padding  
//This variables will be stored in aligned fashion in memory  
char data1 = 0x11;  
int data2 = 0xffffeeee; |  
char data3 = 0x22;  
short data4 = 0abcd;
```

For efficiency, the compiler generates instructions to store variables on their natural size boundary addresses in the memory

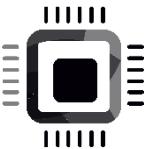
Memory address	Content
0000000000403010 ,	11 char
0000000000403011 ,	0 }
0000000000403012 ,	0 } Padding
0000000000403013 ,	0 }
0000000000403014 ,	EE }
0000000000403015 ,	EE }
0000000000403016 ,	FF } int
0000000000403017 ,	FF }
0000000000403018 ,	22 char
0000000000403019 ,	0 } Padding
000000000040301A ,	CD } short
000000000040301B ,	AB }



Aligned data

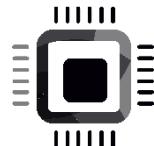
For efficiency, the compiler generates instructions to store variables on their natural size boundary addresses in the memory

Memory address	Content	
000000000061FE04	,	11 char
000000000061FE05	,	0
000000000061FE06	,	0
000000000061FE07	,	0
000000000061FE08	,	EE
000000000061FE09	,	EE
000000000061FE0A	,	FF
000000000061FE0B	,	FF
000000000061FE0C	,	22 char
000000000061FE0D	,	0 Padding
000000000061FE0E	,	CD
000000000061FE0F	,	AB
Total memory consumed by this struct variable = 12		



Aligned data

- By default compiler always stores the data inside the memory in aligned fashion(according to natural size boundary of variables)
- Aligned storage helps to achieve higher performance(in terms of time & code space) using memory load and store instructions
- Aligned storage leads to generate fewer instructions to read data from or write data into memory
- Reduces number of memory hits
- **Aligned data consumes more data memory due to padding**

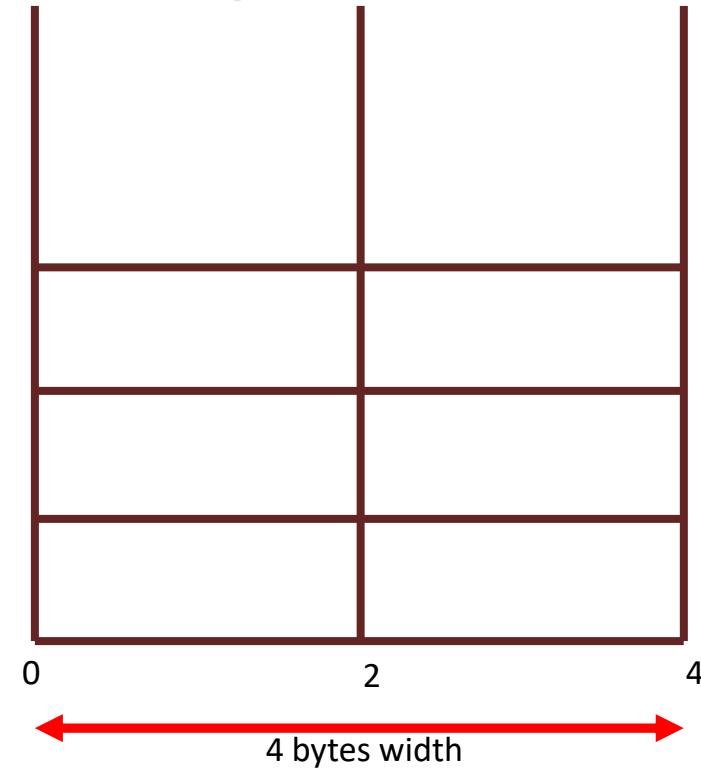


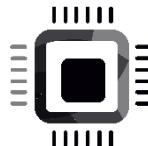
Structure padding

Aligned storage

```
struct data
{
    char  data1 ;
    int   data2 ;
    char  data3 ;
    short data4 ;
};
```

sizeof(struct data) = 12 bytes



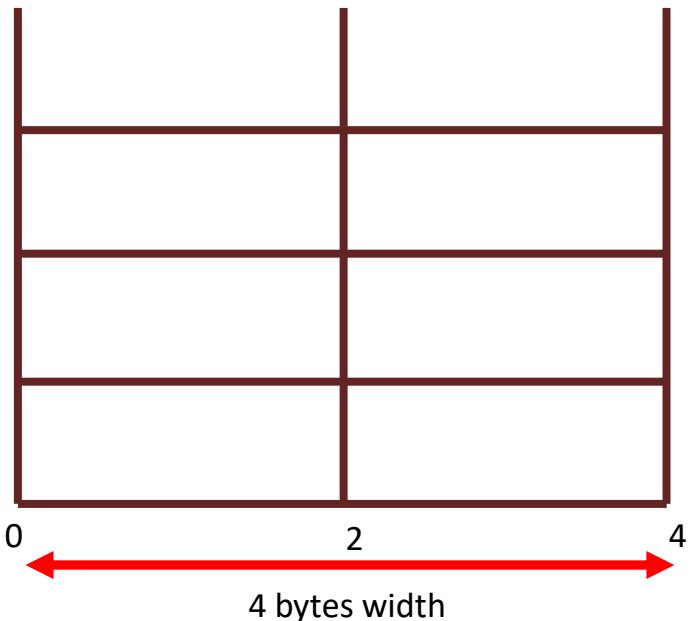


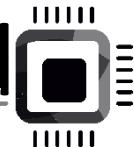
Packed structure (No padding)

```
struct data
{
    char  data1 ;
    int   data2 ;
    char  data3 ;
    short data4 ;
}__attribute__((packed));
```

sizeof(struct data) = 8 bytes

Un-Aligned storage





Assembly code analysis of packed and non-packed structure access

```
arm-none-eabi-size  packed_Vsnonpacked.elf
  text    data    bss    dec    hexfilename
 5112     108   1596   6816
1aa0packed_Vsnonpacked.elf
```

Un-packed

```
arm-none-eabi-size  packed_Vsnonpacked.elf
  text    data    bss    dec    hexfilename
 5144     108   1588   6840
1ab8packed_Vsnonpacked.elf
Finished building: default.size.stdout
```

packed

IDE stm32_examples - 009packed_Vsnonpacked/Src/main.c - STM32CubeIDE

File Edit Source Refactor Navigate Search Project Run Window Help

Quick Access

De... Pr...

009packed_Vsnonpacked.e

009packed_Vsnonpacked

Thread #1 [main] 1 [main() at main.c]

C:/ST/STM32CubeIDE_1

ST-LINK (ST-LINK GDB)

startup_stm... main.c syscalls.c syscalls.c main.c main.c »14

22 struct DataSet

23 {

24 char data1 ;

25 int data2 ;

26 char data3 ;

27 short data4 ;

28 };

29

30

31

32 struct DataSet data ; //this consumes 12 bytes in memory(RAM)

33

34 int main(void)

35 {

36

37 data.data1 = 0xAA;

38 data.data2 = 0xFFFFEEEE;

39 data.data3 = 0x55;

40 data.data4 = 0xA5A5;

41

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);

46

Enter location here

08000292: add r7, sp, #0

37 data.data1 = 0xAA;

08000294: ldr r3, [pc, #76] ; (0x80002e4

08000296: movs r2, #170 ; 0xaa

08000298: strb r2, [r3, #0]

38 data.data2 = 0xFFFFFFFF;

0800029a: ldr r3, [pc, #72] ; (0x80002e4

0800029c: ldr r2, [pc, #72] ; (0x80002e8

0800029e: str r2, [r3, #4]

39 data.data3 = 0x55;

080002a0: ldr r3, [pc, #64] ; (0x80002e4

080002a2: movs r2, #85 ; 0x55

080002a4: strb r2, [r3, #8]

40 data.data4 = 0xA5A5;

080002a6: ldr r3, [pc, #60] ; (0x80002e4

080002a8: movw r2, #42405 ; 0xa5a5

080002ac: strh r2, [r3, #10]

42 printf("data.data1 = %d\n",data.da

080002ae: ldr r3, [pc, #52] ; (0x80002e4

080002b0: ldrb r3, [r3, #0]

080002b2: mov r1, r3

080002b4: ldr r0, [pc, #52] ; (0x80002ec

080002b6: bl 0x8000538 <printf>

43 printf("data.data2 = %d\n",data.da

080002ba: ldr r3, [pc, #40] ; (0x80002e4

080002bc: ldr r3, [r3, #4]

080002be: mov r1, r3

080002c0: ldr r0, [pc, #44] ; (0x80002f0

080002c2: bl 0x8000538 <printf>

Console Register Problem Execute Memory Debug Memory SWV IT...

009packed_Vsnonpacked.elf [STM32 MCU Debugging] ST-LINK (ST-LINK GDB server)



Type here to search



IDE stm32_examples - 009packed_Vsnonpacked/Src/main.c - STM32CubeIDE

File Edit Source Refactor Navigate Search Project Run Window Help

Quick Access

De... Pr...

IDE 009packed_Vsnonpacked.e

009packed_Vsnonpacked

Thread #1 [main] 1 [main() at main.c]

C:/ST/STM32CubeIDE_1

ST-LINK (ST-LINK GDB)

startup_stm... main.c syscalls.c main.c main.c sysmem.c 15

22 struct DataSet

23 {

24 char data1 ;

25 int data2 ;

26 char data3 ;

27 short data4 ;

28 }__attribute__((packed));

29

30

31

32 struct DataSet data ; //this consumes 12 bytes in memory(RAM)

33

34 int main(void)

35 {

36

37 data.data1 = 0xAA;

38 data.data2 = 0xFFFFEEEE;

39 data.data3 = 0x55;

40 data.data4 = 0xA5A5;

41

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);

46

Console 1010 Regist... Proble... Execut... Memor... Debug... Memory SWV IT... 009packed_Vsnonpacked.elf [STM32 MCU Debugging] ST-LINK (ST-LINK GDB server)

(x) Variables Breakpoints Modules Disassembly SFRs

Enter location here

main:

```

08000290: push {r7, lr}
08000292: add r7, sp, #0
37
08000294: ldr r3, [pc, #112] ; (0x8000308
08000296: movs r2, #170 ; 0xaa
08000298: strb r2, [r3, #0]
38
data.data1 = 0xAA;
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
080002bc: strb r2, [r3, #5]
40
data.data4 = 0xA5A5;
080002be: ldr r3, [pc, #72] ; (0x8000308
080002c0: movs r2, #0
080002c2: orn r2, r2, #90 ; 0x5a
080002c6: strb r2, [r3, #6]
080002c8: movs r2, #0

```

IDE 009packed_Vsnonpacked.e 009packed_Vsnonpacked Thread #1 [main] 1 main() at main.c C:/ST/STM32CubeIDE_1 ST-LINK (ST-LINK GDB)

File Edit Source Refactor Navigate Search Project Run Window Help

Variables Breakpoints Modules Disassembly SFRs

```

22 struct DataSet
23 {
24     char data1 ;
25     int data2 ;
26     char data3 ;
27     short data4 ;
28 }__attribute__((packed));
29
30
31
32 struct DataSet data ; //this consumes 12 bytes in memory(RAM)
33
34 int main(void)
35 {
36
37     data.data1 = 0xAA;
38     data.data2 = 0xFFFFEEEE;
39     data.data3 = 0x55;
40     data.data4 = 0xA5A5;
41
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);
46

```

Console Register Problem Execute Memory Debug Memory SWV IT...

009packed_Vsnonpacked.elf [STM32 MCU Debugging] ST-LINK (ST-LINK GDB server)

080002b6: strb r2, [r3, #4] data.data3 = 0x55;
080002b8: ldr r3, [pc, #76] ; (0x8000308
080002ba: movs r2, #85 ; 0x55
080002bc: strb r2, [r3, #5]
080002be: data.data4 = 0xA5A5;
080002c0: ldr r3, [pc, #72] ; (0x8000308
080002c2: movs r2, #0
080002c4: orn r2, r2, #90 ; 0x5a
080002c6: strb r2, [r3, #6]
080002c8: movs r2, #0
080002ca: orn r2, r2, #90 ; 0x5a
080002ce: strb r2, [r3, #7]
080002d0: printf("data.data1 = %d\n",data.da
080002d2: ldr r3, [pc, #52] ; (0x8000308
080002d4: ldrb r3, [r3, #0]
080002d6: mov r1, r3
080002d8: ldr r0, [pc, #52] ; (0x800030c
080002dc: bl 0x8000558 <printf>
080002de: printf("data.data2 = %d\n",data.da
080002e0: ldr r3, [pc, #40] ; (0x8000308
080002e2: ldr.w r3, [r3, #1]
080002e4: mov r1, r3
080002e6: ldr r0, [pc, #40] ; (0x8000310
080002ea: bl 0x8000558 <printf>
080002ec: printf("data.data3 = %d\n",data.da
080002ee: ldr r3, [pc, #28] ; (0x8000308
080002f0: ldrb r3, [r3, #5]
080002f2: mov r1, r3
080002f4: ldr r0, [pc, #28] ; (0x8000310
080002f6: bl 0x8000558 <printf>



Typedef with structure

struct CarModel

{

```
unsigned int carNumber;
uint32_t    carPrice;
uint16_t    carMaxSpeed;
float       carWeight;
```

```
},  
struct CarModel carBMW, carFord;
```

typedef is used to give an alias name to primitive and user defined data types

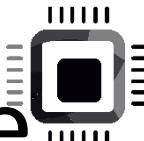
typedef struct

{

```
unsigned int carNumber;
uint32_t    carPrice;
uint16_t    carMaxSpeed;
float       carWeight;
```

} CarModel_t;

CarModel_t carBMW, carFord;

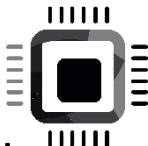


Some more points about structure

- A structure type cannot contain itself as a member

```
struct CarModel
```

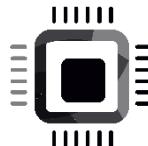
```
{  
    unsigned int carNumber;  
    uint32_t    carPrice;  
    uint16_t    carMaxSpeed;  
    float       carWeight;  
    struct CarModel carBWM; //Not allowed  
};
```



- structure types can contain pointers to their own type.
- Such self-referential structures are used in implementing linked lists and binary trees

```
struct CarModel
```

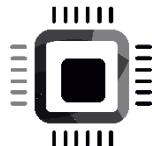
```
{  
    unsigned int carNumber;  
    uint32_t    carPrice;  
    uint16_t    carMaxSpeed;  
    float       carWeight;  
    struct CarModel *pcarBWM; //allowed  
};
```



Nested structure (Structure inside a structure)

```
struct Data
{
    char data1 ;
    int data2 ;
    char data3 ;
    short data4 ;
    struct
    {
        char data5;
        int data6;
    }moreData;
};
```

```
struct CarModel
{
    unsigned int carNumber;
    uint32_t carPrice;
    uint16_t carMaxSpeed;
    float carWeight;
    struct
    {
        float temperature;
        float airPressure;
        int fuel;
    }carParameters;
};
```



Structures and pointers

- Creating pointer variables of a structure
- Reading and writing data with member elements using structure pointers

Storage of a structure in memory

```
struct DataSet
{
    char data1;
    int data2;
    char data3;
    short data4;
};

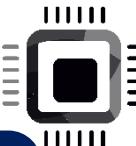
int main(void)
{
    struct DataSet data;

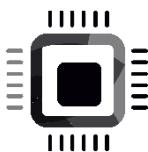
    data.data1 = 0x11;
    data.data2 = 0xFFFFEEEE;
    data.data3 = 0x22;
    data.data4 = 0xABCD;
}
```

Base address of the structure in memory also happens to be the address of the first member element

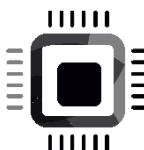
Memory address	Content
000000000061FE04 ,	11
000000000061FE05 ,	0
000000000061FE06 ,	0
000000000061FE07 ,	0
000000000061FE08 ,	EE
000000000061FE09 ,	EE
000000000061FE0A ,	FF
000000000061FE0B ,	FF
000000000061FE0C ,	22
000000000061FE0D ,	0
000000000061FE0E ,	CD
000000000061FE0F ,	AB

Total memory consumed by this struct variable = 12





Let's say you have been given with the base address of a structure variable and asked to change the member element values what would you do ?



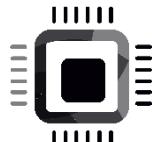
```
//pData is a pointer variable of type struct DataSet *
struct DataSet *pData;

//pData now holds the address of structure variable data
pData = &data;

//changing the value of data1 member element
pData->data1 = 0x55;
```

A red bracket underlines the code `pData->data1`. A blue arrow points from the end of the bracket down to the equivalent expression `*(address_of_first_member_element_data1)= 0x55;`.

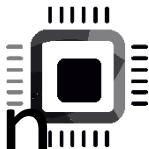
`*(address_of_first_member_element_data1)= 0x55;`



Accessing Structure Members

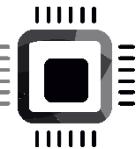
Use dot (.) operator when you use structure variable of non pointer type

Use arrow (->) operator when you use structure variable of pointer type

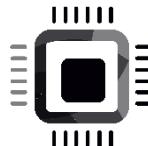


Passing structure variable to a function

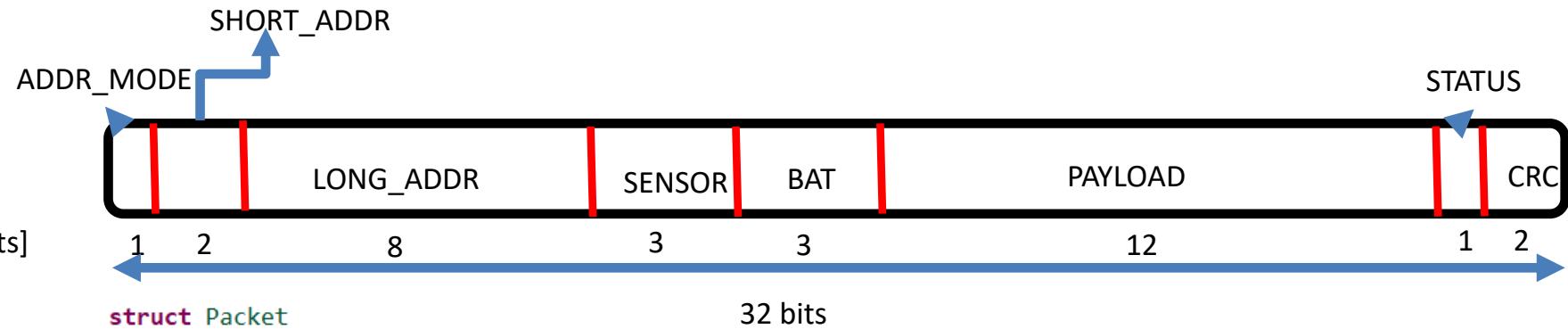
- Pass by value
- Pass by reference (pointer)



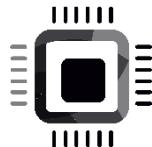
Structure serialization and address analysis



Write a program to decode a given 32bit packet information and print the values of different fields. Create a structure with member elements as packet fields as shown below

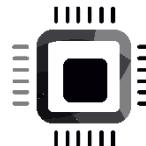


```
struct Packet
{
    crc;
    status;
    payload;
    bat;
    sensor;
    longAddr;
    shortAddr;
    addrMode;
};
```



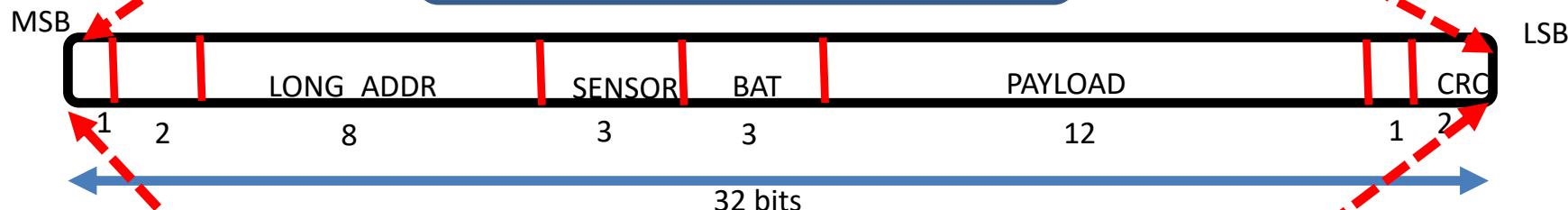
Exercise

- Re-write the LED toggle application using structure and bitfields. You have to create a structure for every peripheral register you use in this program.
- Access the structure member elements to configure and access the peripheral registers.

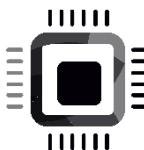


`uint32_t packetValue;`

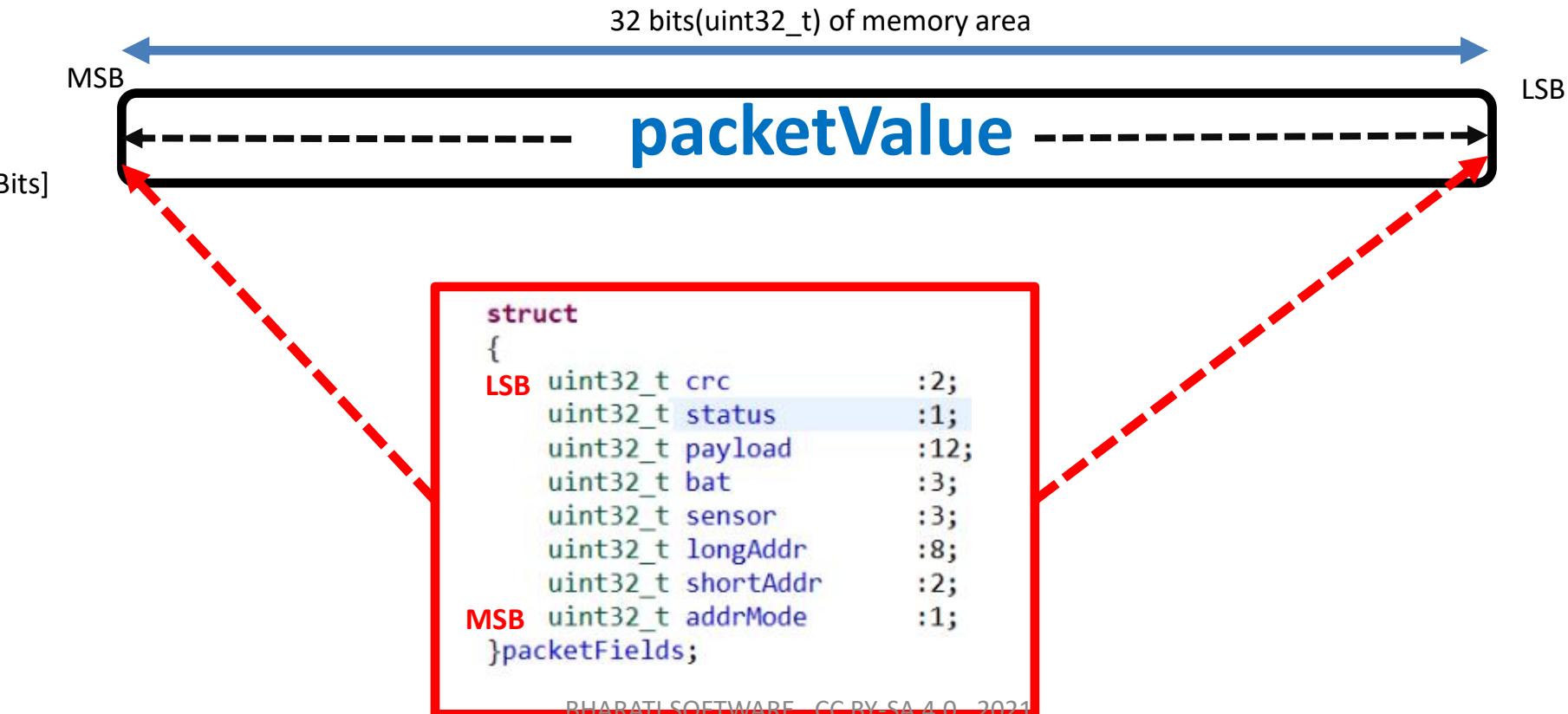
In union, all the member elements refer to the same memory area.

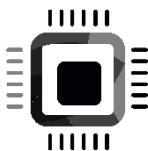


```
struct
{
    LSB
    uint32_t crc :2;
    uint32_t status :1;
    uint32_t payload :12;
    uint32_t bat :3;
    uint32_t sensor :3;
    uint32_t longAddr :8;
    uint32_t shortAddr :2;
    uint32_t addrMode MSB :1;
}packetFields; I
```

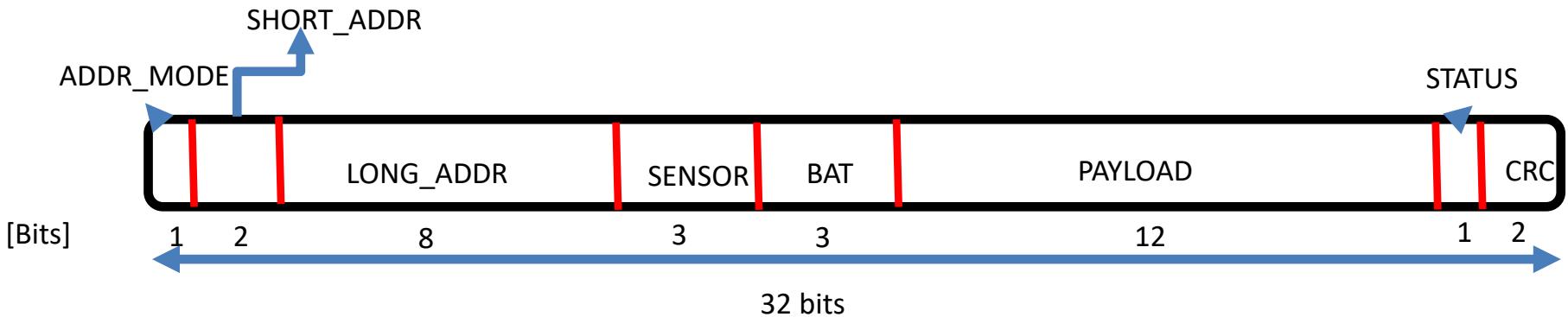


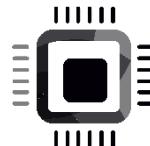
In union, all the member elements refer to the same memory area.





Bit-fields





RCC AHB1 peripheral clock enable register (RCC_AHB1ENR)

Address offset: 0x30

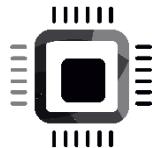
Reset value: 0x0010 0000

Access: no wait state, word, half-word and byte access.

	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	
Reser- ved	OTGH S ULPIE N	OTGH SEN	ETHM ACPTP EN	ETHM ACRXE N	ETHM ACTXE N	ETHMA CEN	Reserved	DMA2E N	DMA1E N	CCMDAT ARAMEN	Res.	BKPSR AMEN	Reserved	rw	rw	rw	
	rw	rw	rw	rw	rw	rw											
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	0	
Reserved			CRCE N	Reserved			GPIOE N	GPIOH EN	GPIOG EN	GPIOF EN	GPIOEEN	GPIOD EN	GPIOC EN	GPIO BEN	GPIO AEN		
			rw				rw	rw	rw	rw	rw	rw	rw	rw	rw		

So, can you convert this register information into a structure with bit fields ?

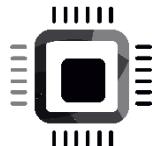
Creating bit-field structure for peripheral registers



```
typedef struct  
{
```

?

```
}RCC_AHB1ENR_t;
```



Create different Bit field structures for different peripheral registers you use in LED toggle application.

```
typedef struct  
{
```

?

```
}RCC_AHB1ENR_t;
```

```
typedef struct  
{
```

?

```
}GPIOx_MODE_t;
```

```
typedef struct  
{
```

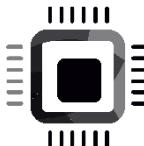
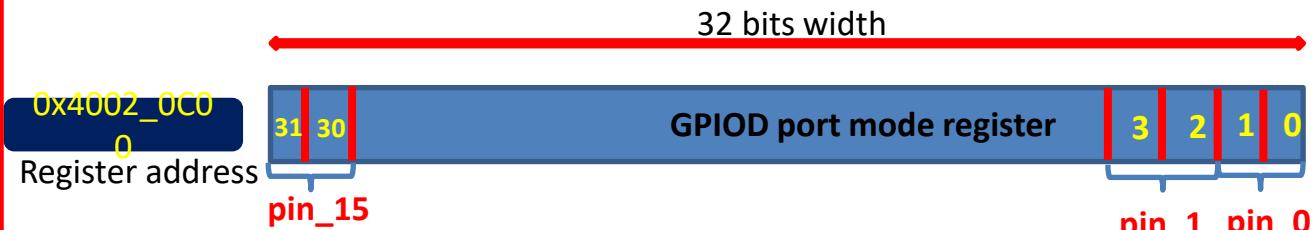
?

```
}GPIOx_ODR_t;
```

```
{
    uint32_t pin_0:2;
    uint32_t pin_1:2;
    uint32_t pin_2:2;
    uint32_t pin_3:2;
    uint32_t pin_4:2;
    uint32_t pin_5:2;
    uint32_t pin_6:2;
    uint32_t pin_7:2;
    uint32_t pin_8:2;
    uint32_t pin_9:2;
    uint32_t pin_10:2;
    uint32_t pin_11:2;
    uint32_t pin_12:2;
    uint32_t pin_13:2;
    uint32_t pin_14:2;
    uint32_t pin_15:2;
}GPIOx_MODE_t;
```

GPIOx_MODE_t *pGpiodMode;

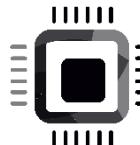
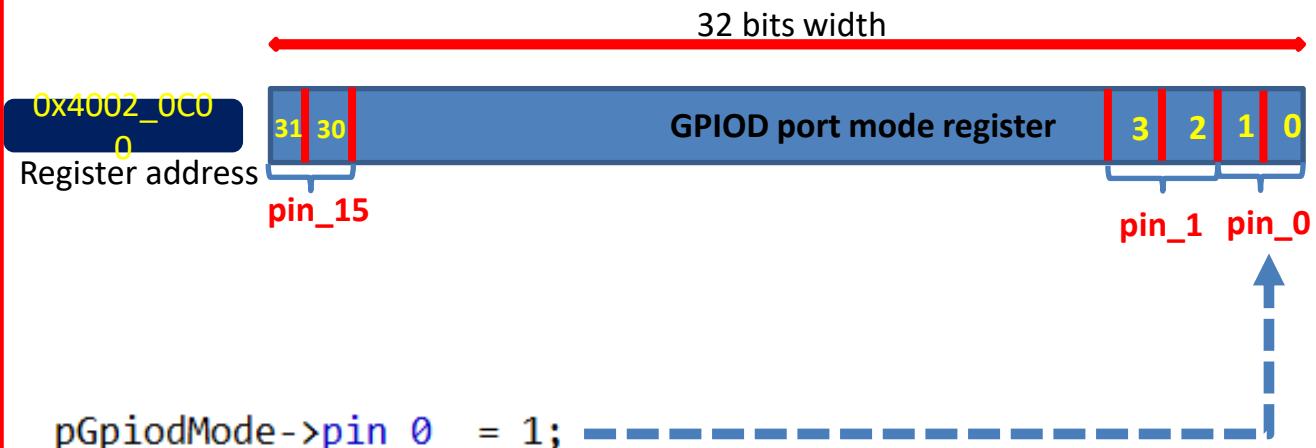
pGpiodMode = (GPIOx_MODE_t*) 0x40020C00;



```
{
    uint32_t pin_0:2;
    uint32_t pin_1:2;
    uint32_t pin_2:2;
    uint32_t pin_3:2;
    uint32_t pin_4:2;
    uint32_t pin_5:2;
    uint32_t pin_6:2;
    uint32_t pin_7:2;
    uint32_t pin_8:2;
    uint32_t pin_9:2;
    uint32_t pin_10:2;
    uint32_t pin_11:2;
    uint32_t pin_12:2;
    uint32_t pin_13:2;
    uint32_t pin_14:2;
    uint32_t pin_15:2;
}GPIOx_MODE_t;
```

GPIOx_MODE_t *pGpiodMode;

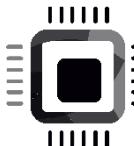
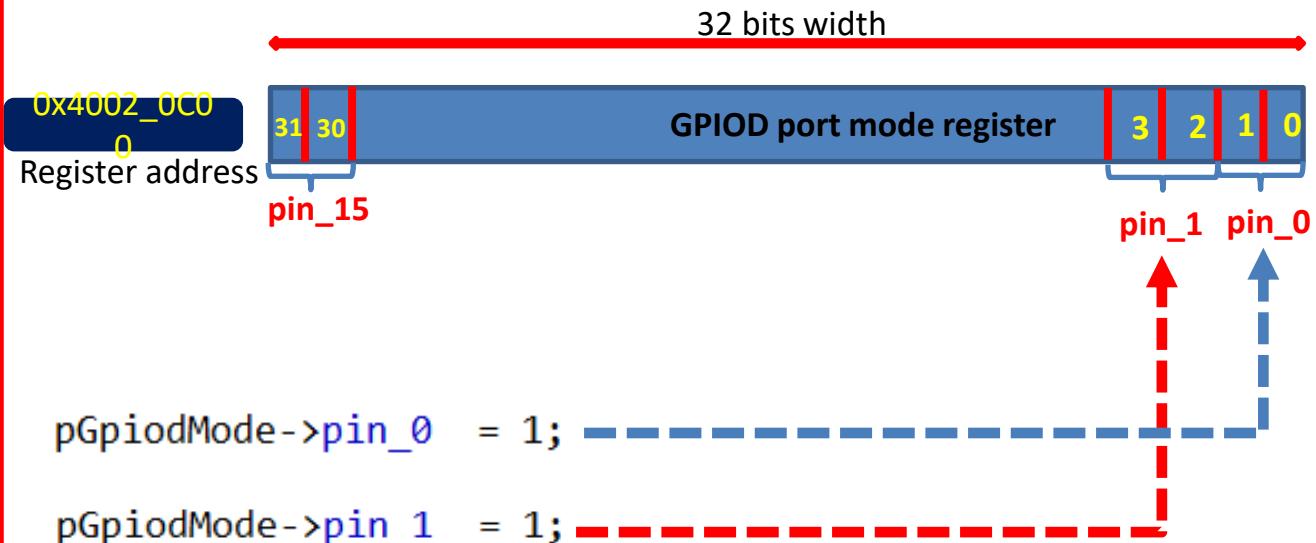
pGpiodMode = (GPIOx_MODE_t*) 0x40020C00;



```
{
    uint32_t pin_0:2;
    uint32_t pin_1:2;
    uint32_t pin_2:2;
    uint32_t pin_3:2;
    uint32_t pin_4:2;
    uint32_t pin_5:2;
    uint32_t pin_6:2;
    uint32_t pin_7:2;
    uint32_t pin_8:2;
    uint32_t pin_9:2;
    uint32_t pin_10:2;
    uint32_t pin_11:2;
    uint32_t pin_12:2;
    uint32_t pin_13:2;
    uint32_t pin_14:2;
    uint32_t pin_15:2;
}GPIOx_MODE_t;
```

GPIOx_MODE_t *pGpiodMode;

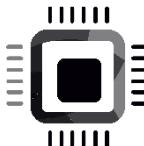
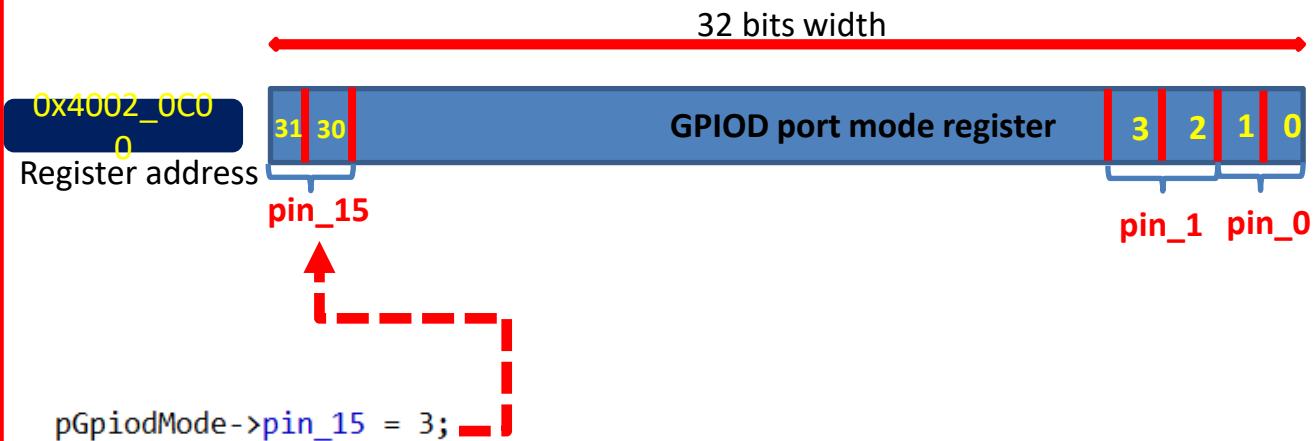
pGpiodMode = (GPIOx_MODE_t*) 0x40020C00;



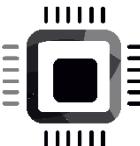
```
{
    uint32_t pin_0:2;
    uint32_t pin_1:2;
    uint32_t pin_2:2;
    uint32_t pin_3:2;
    uint32_t pin_4:2;
    uint32_t pin_5:2;
    uint32_t pin_6:2;
    uint32_t pin_7:2;
    uint32_t pin_8:2;
    uint32_t pin_9:2;
    uint32_t pin_10:2;
    uint32_t pin_11:2;
    uint32_t pin_12:2;
    uint32_t pin_13:2;
    uint32_t pin_14:2;
    uint32_t pin_15:2;
}GPIOx_MODE_t;
```

GPIOx_MODE_t *pGpiodMode;

pGpiodMode = (GPIOx_MODE_t*) 0x40020C00;

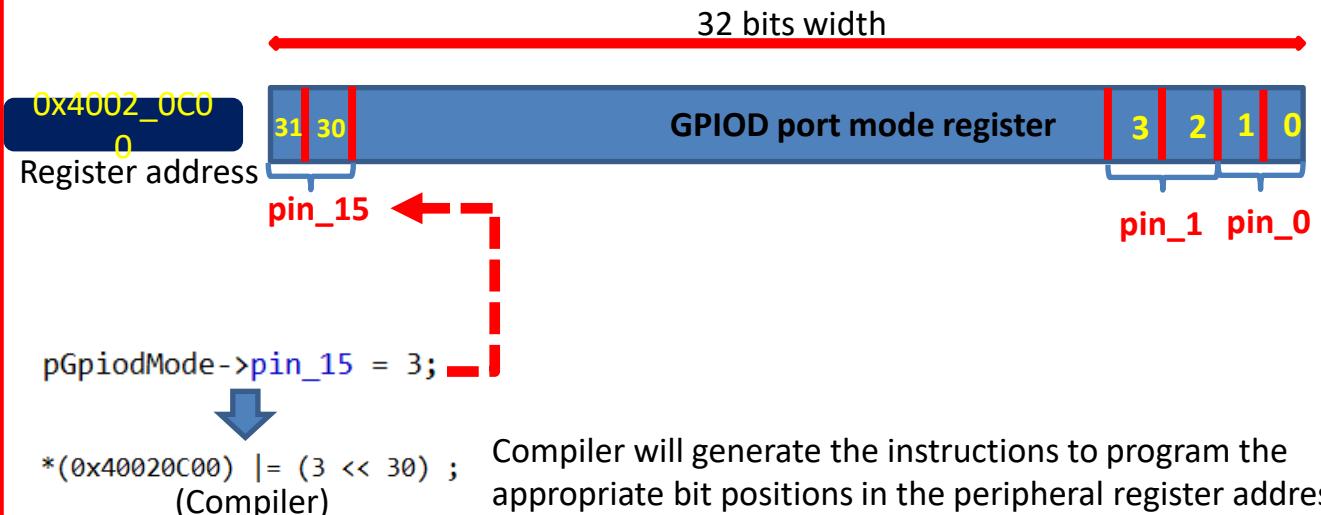


```
{
    uint32_t pin_0:2;
    uint32_t pin_1:2;
    uint32_t pin_2:2;
    uint32_t pin_3:2;
    uint32_t pin_4:2;
    uint32_t pin_5:2;
    uint32_t pin_6:2;
    uint32_t pin_7:2;
    uint32_t pin_8:2;
    uint32_t pin_9:2;
    uint32_t pin_10:2;
    uint32_t pin_11:2;
    uint32_t pin_12:2;
    uint32_t pin_13:2;
    uint32_t pin_14:2;
    uint32_t pin_15:2;
}GPIOx_MODE_t;
```

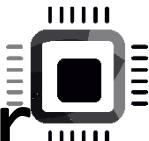


GPIOx_MODE_t *pGpiodMode;

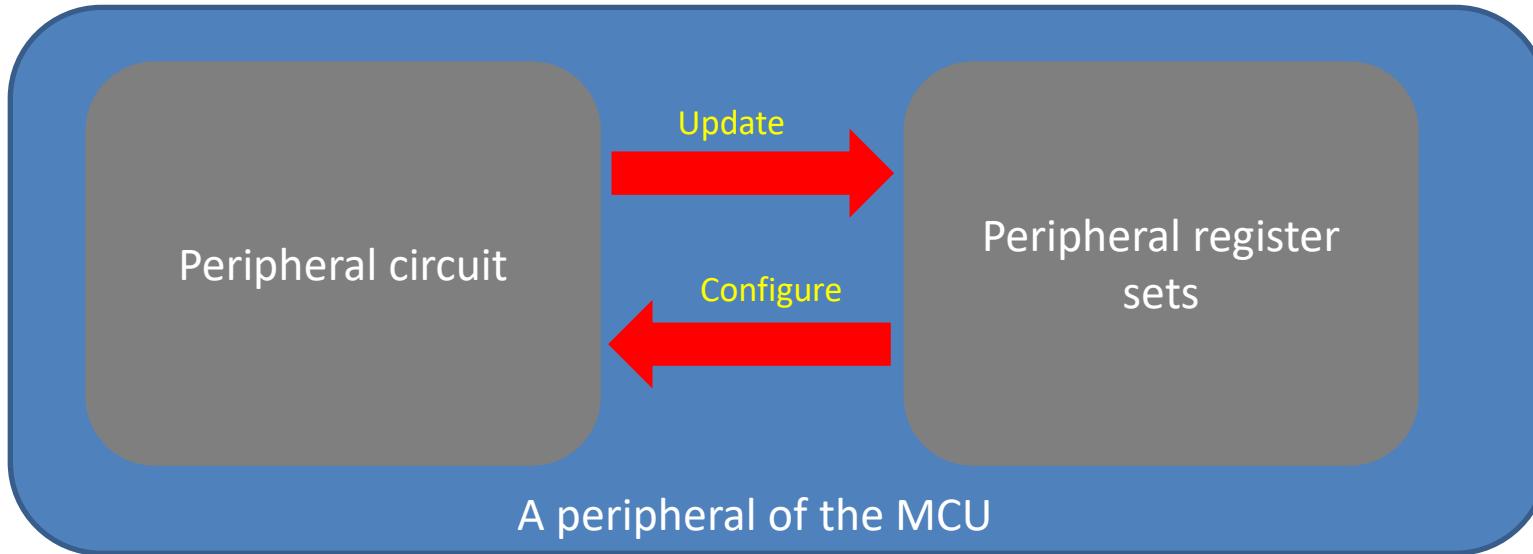
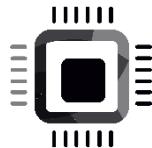
pGpiodMode = (GPIOx_MODE_t*) 0x40020C00;



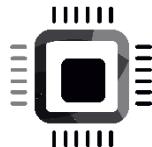
Volatile data and structure pointer



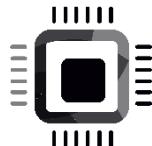
Creating a structure for the MCU peripheral



Creating a structure for the MCU peripheral

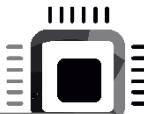


Let's create a generic structure for a particular peripheral and keep all individual peripheral register structures as member elements.



GPIO Peripherals in STM32

- STM32F407 has 11 GPIO peripherals
- GPIOA, GPIOB GPIOK (GPIO x , $x = A..I/J/K$)
- Each GPIO peripheral is used to control 16 different pins of the port
 - E.g. : GPIOA controls PORT-A (pins PA0 to PA15)



GPIOA

(Register base address of

GPIOA

0x40020000

0

- GPIOA port mode register
- GPIOA Port output type register
- GPIOA Port output speed register
- GPIOA port pull-up/pull-down register
- GPIOA port input data register
- GPIOA port output data register
- GPIOA port bit set/reset register
- GPIOA port configuration lock register
- GPIOA port alternate function low register
- GPIOA port alternate function high register

GPIOB

(Register base address of **GPIOB**)

0x40020400

0

- GPIOB port mode register
- GPIOB Port output type register
- GPIOB Port output speed register
- GPIOB port pull-up/pull-down register
- GPIOB port input data register
- GPIOB port output data register
- GPIOB port bit set/reset register
- GPIOB port configuration lock register
- GPIOB port alternate function low register
- GPIOB port alternate function high register

GPIOC

(Register base address of **GPIOC**)

0x40020800

0

- GPIOC port mode register
- GPIOC Port output type register
- GPIOC Port output speed register
- GPIOC port pull-up/pull-down register
- GPIOC port input data register
- GPIOC port output data register
- GPIOC port bit set/reset register
- GPIOC port configuration lock register
- GPIOC port alternate function low register
- GPIOC port alternate function high register

GPIOD

(Register base address of

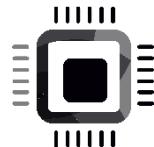
GPIOD

0x40020C00

0

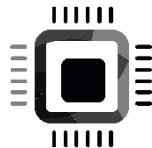
- GPIOD port mode register
- GPIOD port output type register
- GPIOD port output speed register
- GPIOD port pull-up/pull-down register
- GPIOD port input data register
- GPIOD port output data register
- GPIOD port bit set/reset register
- GPIOD port configuration lock register
- GPIOD alternate function low register
- GPIOD alternate function high register

Every peripheral has its own register sets falling in specific locations of the processor memory map



Exercise

- Create a generic structure for the GPIO peripherals, which can be used to configure any GPIO peripheral.
- Keep individual bit field structures created in the previous lecture as member elements of the above mentioned generic structure.
- Re-Write the LED toggle application with the above changes.



Generic structure for GPIOx

typedef struct

{

?

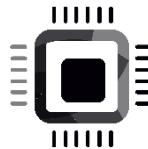
}GPIOx_t;

Note:

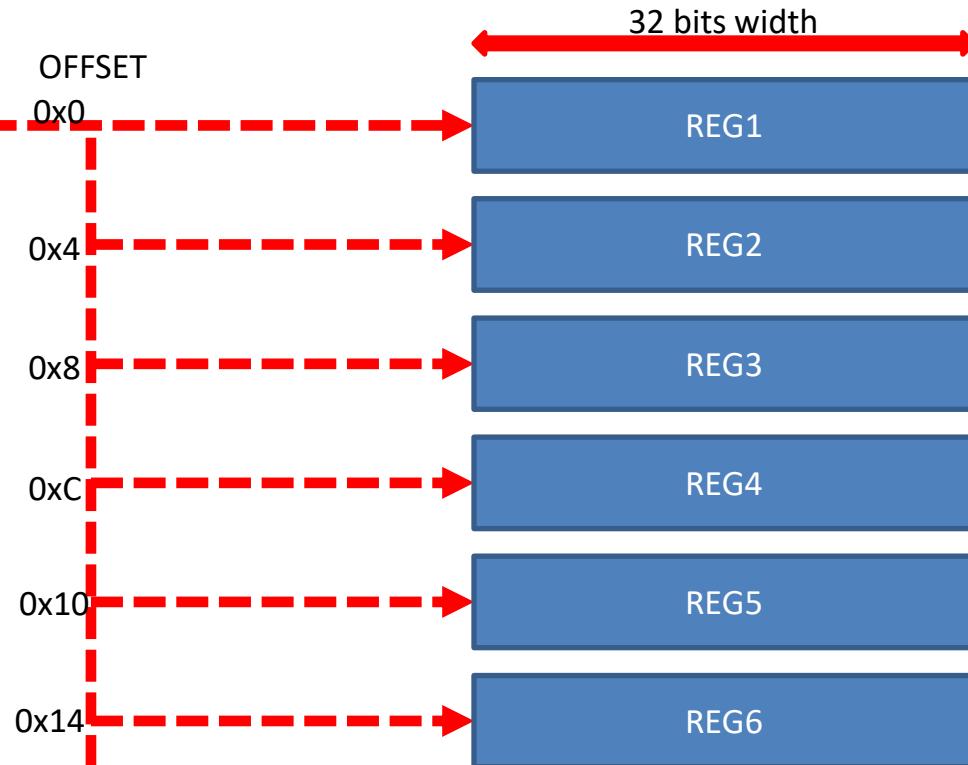
While writing member elements into this structure, give attention to offsets of different registers.

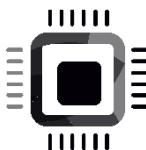
Not maintaining the proper offset will end up programming wrong addresses

Register base address of a peripheral

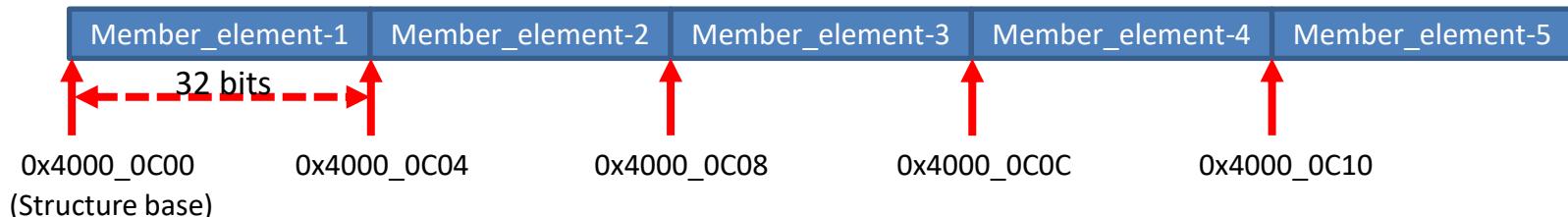


0x400X_XXXX

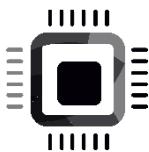




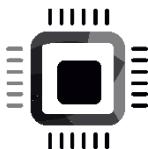
Memory locations



Address of the Member-element-1 : 0x4000_0C00 Corresponds to REG1
Address of the Member-element-2 : 0x4000_0C04 Corresponds to REG2
Address of the Member-element-3 : 0x4000_0C08 Corresponds to REG3
Address of the Member-element-4 : 0x4000_0C0C Corresponds to REG3
so on . . .



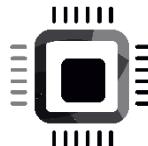
unions



unions

- A *union* in ‘C’ is similar to a structure except that all of its members start at the same location in memory.
- A union variable can represent the value of only one of its members at a time.

Difference between union and structure



Structure memory allocation

0xE00



0xE01



0xE02



0xE03



0xE04



0xE05



0xE06



0xE07



union memory allocation

0xE00



0xE01



0xE02



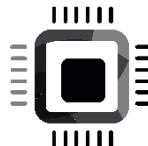
0xE03



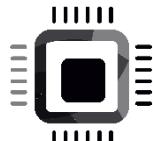
```
struct address
{
    uint16_t shortAddr;
    uint32_t longAddr;
};
```

```
union address
{
    uint16_t shortAddr;
    uint32_t longAddr;
};
```

Applicability of unions in Embedded System code

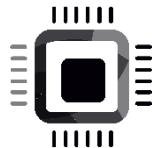


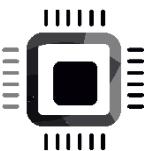
1. Bit extraction
2. Storing mutually exclusive data thus saving memory



Unions and bit extraction technique

Unions to store mutually exclusive data



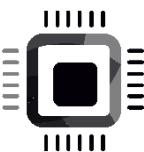


```
typedef struct
{
    union
    {
        uint16    shortAddr;
        ZLongAddr_t extAddr;
    } addr;
    byte addrMode;
} zAddrType_t;
```

```
/* Extended address */
typedef uint8 sAddrExt_t[SADDR_EXT_LEN];

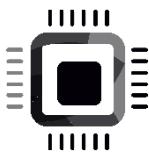
/* Combined short/extended device address */
typedef struct
{
    union
    {
        uint16    shortAddr; /* Short address */
        sAddrExt_t extAddr; /* Extended address */
    } addr;
    uint8     addrMode;   /* Address mode */
} sAddr_t;
```

Some example use cases of unions

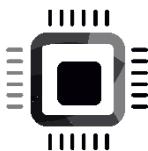


```
// Scan Request/Response "ZigBee information" field bitmap
typedef struct
{
    unsigned int logicalType:2;
    unsigned int rxOnWhenIdle:1;
    unsigned int reserved:5;
} zInfoBits_t;

// Scan Request/Response "ZigBee information" field
typedef union
{
    zInfoBits_t zInfoBits;
    uint8 zInfoByte;
} zInfo_t;
```

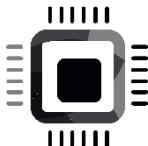


Optimization



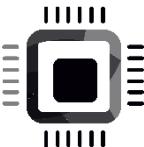
Exercise

Modify the LED Toggle program by introducing structure and bit fields to configure and access the memory mapped peripheral registers.



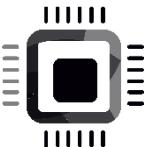
Program optimization

- Optimization is a series of actions taken by the compiler on your program's code generation process to reduce
 - Number of instructions (code space optimization)
 - Memory access time (time space optimization)
 - Power consumption
- By default, the compiler ~~doesn't~~ invoke any optimization on your program
- You can enable the optimization using compiler flags



GCC Compiler flags to enable optimization

- -O0
 - No optimization .
 - Not recommended for productions if you have limited code and ram space
 - Has fastest compilation time
 - This is debugging friendly and used during development
 - A code which works with –O0 optimization may not work with –O0+ optimization levels. Code needs to verified again.
- -O1
 - Moderate optimization to decrease memory access time and code space



GCC Compiler flags to enable optimization

-O2

- Full optimization
- Slow compilation time
- Not debugging friendly

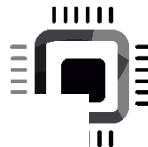
You need to work with different optimization levels to find out what works for you.

For all code exercise we do in this course, we consider -O0 optimization level.

-O3

- Full optimization of -O2 + some more aggressive optimization steps will be taken by the compiler.
- Slowest compilation time
- May cause bugs in the program
- Not debugging friendly

Interfacing projects



Code from scratch using the concepts of ,

- Pointers
- Memory mapped IOs
- Bitwise operators
- Structures and bit fields
- const and volatile type qualifiers

