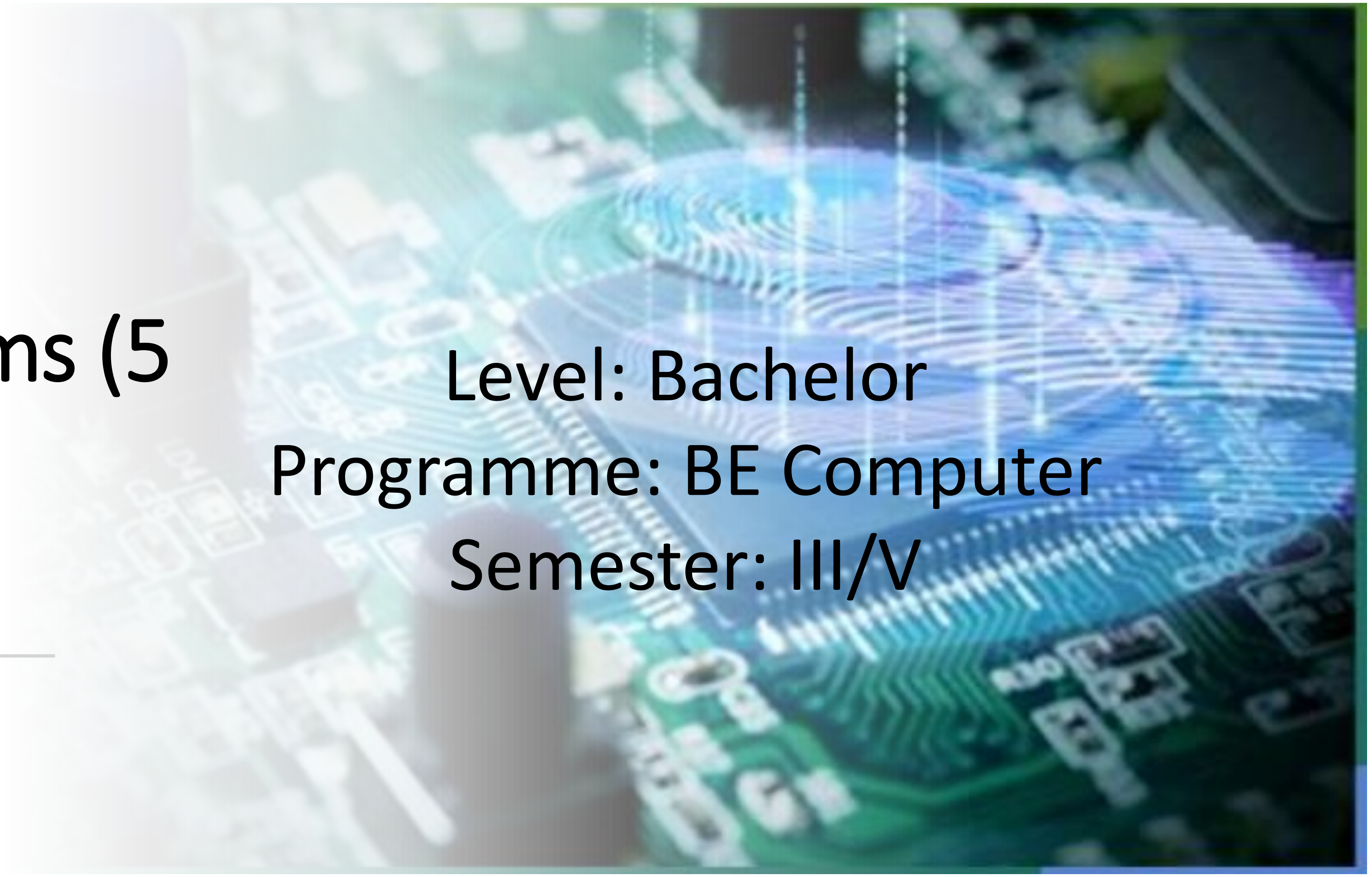
Embedded Chapter 2. Programming for Embedded systems (5 hrs)



Level: Bachelor
Programme: BE Computer
Semester: III/V

Ishwor Koirala, Ph.D.

Instructor

Outline

Introduction

Books

Course Contents

1. Introduction to Embedded system (3 hrs)
2. **Programming for Embedded systems (5 hrs)**
3. Real-time operating systems (RTOS) (5 hrs)
4. Embedded System Design using VHDL (5 hrs)
5. Communications Protocals (3 hrs)
6. Pheripherals and Interfacing (4 hrs)
7. Internet of Things (IoT)and Embedded system (3hrs)

Overview of AVR Architecture

(Micro) processor, computer, controller

CPU: is a unit that fetches and processes a set of general-purpose instructions.

Microprocessor: is a CPU on a single chip. It may also have other units (e.g. caches, floating point processing arithmetic faster processing of instructions. (Intel 4004)

Microcomputer: when a microprocessor + I/O + memory+ etc are put together to form a small computer for applications like data collection, or control application.

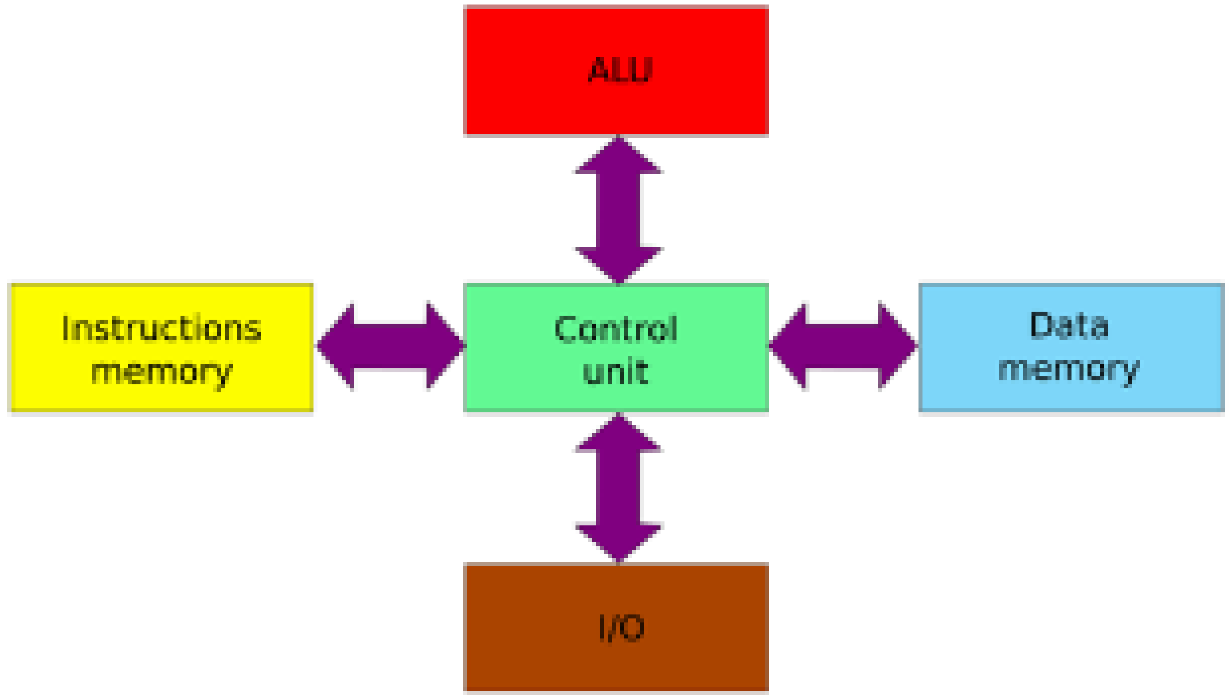
Microcontroller (MCU): a microcomputer on a single chip. It brings together the microprocessor core and a rich collection of peripherals and I/O capability.

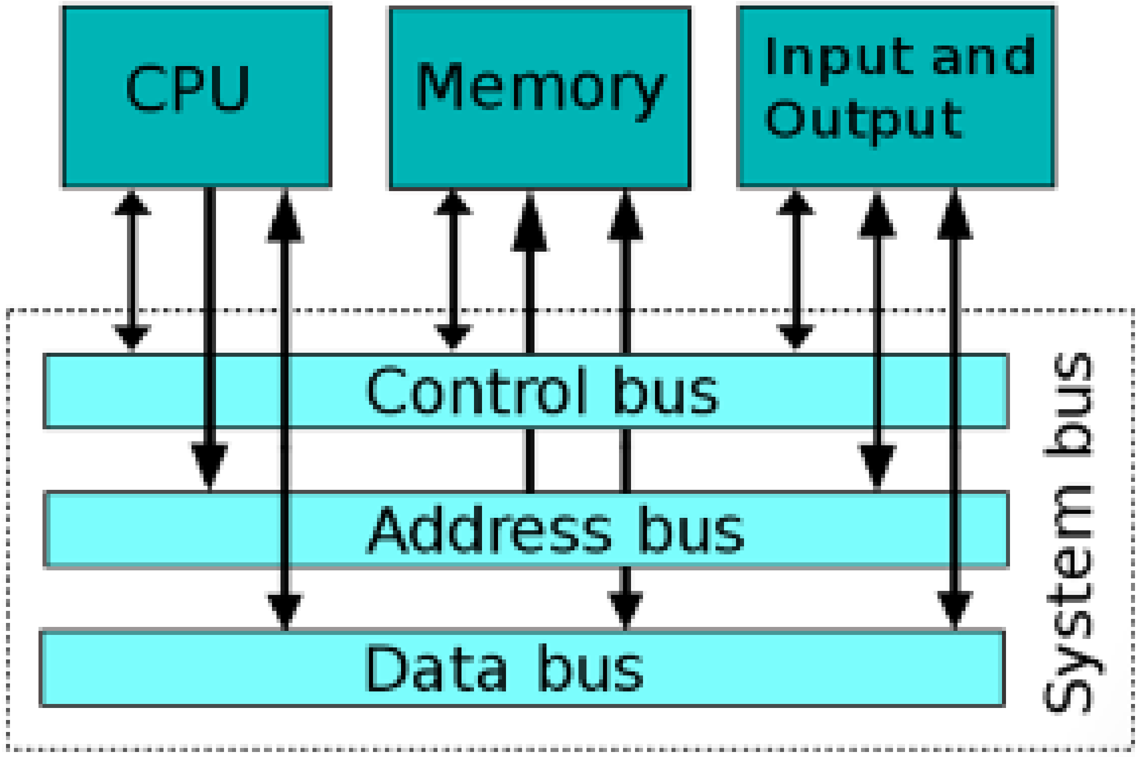
Overview of AVR Architecture

- AVR microcontrollers use a modified Harvard architecture, which means that program and data are stored in separate memory systems:
- **Program memory**: In-System Reprogrammable Flash memory. Instructions in the program memory are executed using single-level pipelining, which allows instructions to be executed in every clock cycle.
- **Data memory**: Static Random Access Memory (SRAM).
- **Address buses**: The Flash Program Memory space is on a separate address bus than the SRAM.
- **Data buses**: There are two data buses, one that can access all data and the In/Out data bus with limited access to a small section of memory.

https://eclipse.umbc.edu/robucci/cmpe311/Lectures/L02-AVR_Archetecture/

Overview of AVR Architecture





Overview of AVR Architecture

- RISC Harvard Architecture
- RISC not CISC
- Harvard not Von Neumann
  - ➢ Separate Program and Data memory
  - ➢ On-chip flash memory-->program memory
    - ✓ execute in-place
  - ➢ On-chip data memory (RAM and EEPROM)
- 32 x 8 General purpose registers
- On-chip programmable timers
- Internal and external interrupt sources
- Programmable watchdog timer
- On-chip RC clock oscillator (more on clock later)
- Variety of I/O, Programmable I/O lines

2.2 Embedded C Programming for Microcontroller

Introduction to C for Embedded Systems

- ✓ Embedded C language is used to develop microcontroller-based applications.
- ✓ Embedded C is an extension to the C programming language including different features such as addressing I/O, fixed-point arithmetic, multiple-memory addressing, etc.
- ✓ In embedded C language, specific compilers are used.

Data types, Control Structures, and Pointers

- In an embedded C programming language, we can place comments in our code which helps the reader to understand the code easily.

C=a+b; /* add two variables whose value is stored in another variable C*/

Single Line Comment

➢ These comments begin with a double slash (//) and it can be located anywhere within the programming language.

Multi-Line Comment

➢ begin with a single slash (/) & an asterisk (/*) in the programming languages which explains a block of code

Directives of Processor

➢ The lines included within the program code are called preprocessor directives which can be followed through a hash symbol (#).

➢ These lines are the preprocessor directives but not programmed statements.

Data types, Control Structures, and Pointers

- #include

```
#include<reg51.h>
Sbit LED = P2^3;
Main();
{
LED = 0x0ff
Delay();
LED=0x00;
}
```

#define

```
#include<reg51.h>
#define LED P0
Main();
{
LED = 0x0ff
Delay();
LED=0x00;
}
```

- Here, the #include directive is generally used to comprise standard libraries like study and. h is used to allow I/O functions using the library of ‘C’. The #define directive usually used to describe the series of variables & allocates the values by executing the process within a particular instruction like macros.

Data types, Control Structures, and Pointers

sbit

Data type, used to access a single bit within an SFR register.

The syntax for this data type is : sbit variable name = SFR bit ;

Example: sbit a=P2^1;

If we assign p2.1 as ‘a’ variable, then we can use ‘a’ instead of p2.1 anywhere in the program, which reduces the complexity of the program.

Bit

This type of data type is mainly used for allowing the bit addressable memory of random access memory like 20h to 2fh.

The syntax of this data type is : name of bit variable;

Example: bit c;

It is a bit series setting within a small data region that is mainly used with the help of a program to memorize something.

Data types, Control Structures, and Pointers

SFR Register

The SFR stands for Special Function Register.

In 8051 microcontroller, it includes the RAM memory with 256 bytes, which is divided into two main elements: the first element of 128 bytes is mainly utilized for storing the data whereas the other element of 128 bytes is mainly utilized to SFR registers.

All the peripheral devices such as timers, counters & I/O ports are stored within the SFR register & every element includes a single address.

Data types, Control Structures, and Pointers

SFR

This kind of data type is used to obtain the peripheral ports of the SFR register through an additional name. So, the declaration of all the SFR registers can be done in capital letters.

The syntax of this data type is: SFR variable name = SFR address for SFR register;

Example: SFR port0 = 0×80;

If we allocate 0×80 like ‘port0’, after that we can utilize 0×80 in place of port0 wherever in the programming language to decrease the difficulty of the program.

Data types, Control Structures, and Pointers

Global Variables

When the variable is declared before the key function is known as the global variable. This variable can be allowed on any function within the program. The global variable’s life span mainly depends on the programming until it reaches an end.

```
#include<reg51.h>
Unsigned int a, c =10;
Main()
{
……………
…………..
}
```

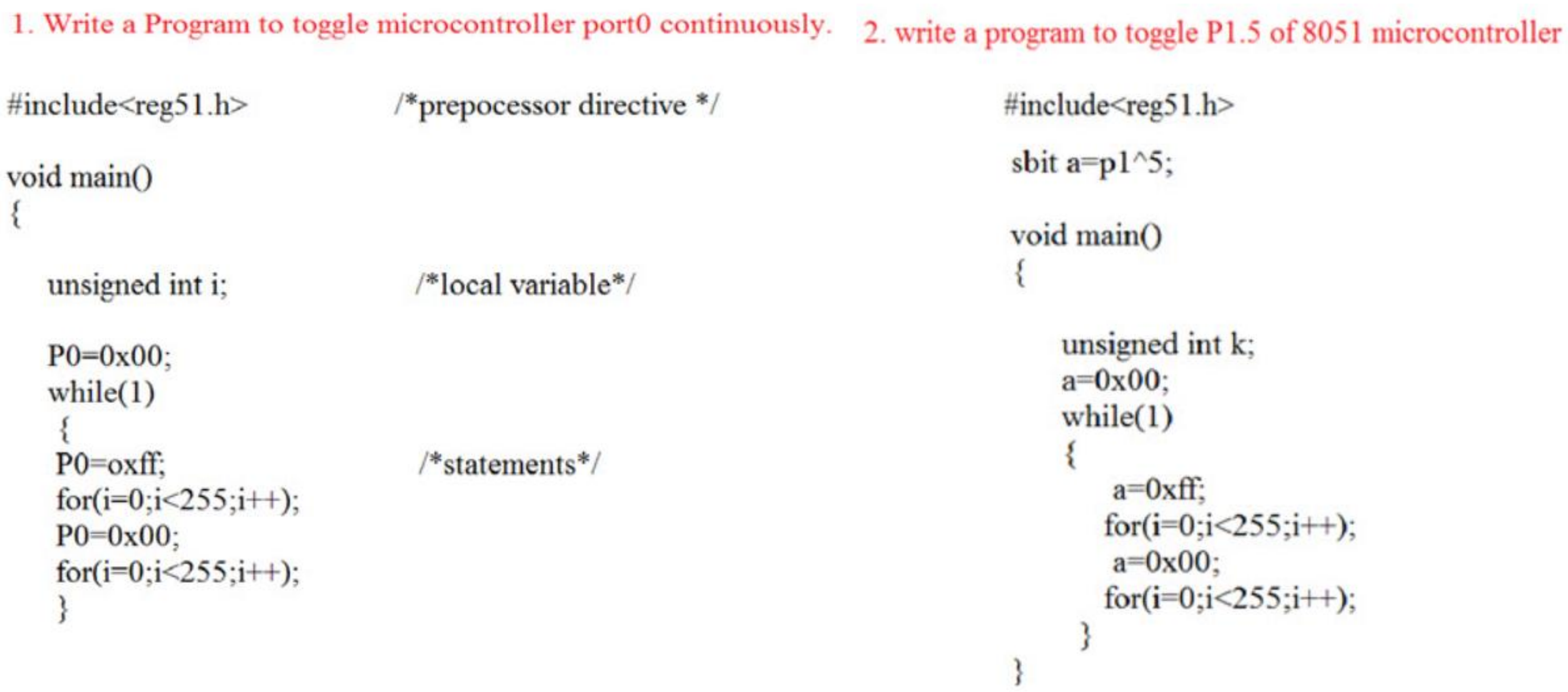
Difference between C Programs and Embedded C Program

C Language	**Embedded C Language**
Generally, this language is used to develop desktop-based applications	Embedded C language is used to develop microcontroller-based applications.
C language is not an extension to any programming language, but a general-purpose programming language	Embedded C is an extension to the C programming language including different features such as addressing I/O, fixed-point arithmetic, multiple-memory addressing, etc.
It processes native development in nature	It processes cross development in nature
It is independent for hardware architecture	It depends on the hardware architecture of the microcontroller & other devices
The compilers of C language depends on the operating system	Embedded C compilers are OS independent

Difference between C Programs and Embedded C Program Contd..

In C language, the standard compilers are used for executing a program	In embedded C language, specific compilers are used.
The popular compilers used in this language are GCC, Borland turbo C, Intel C++, etc	The popular compilers used in this language are Keil, BiPOM Electronics & green hill
The format of C language is free-format	Its format mainly depends on the kind of microprocessor used.
Optimization of this language is normal	Optimization of this language is a high level
It is very easy to modify & read	It is not easy to modify & read
Bug fixing is easy	Bug fixing of this language is complicated

Program



1. Write a Program to toggle microcontroller port0 continuously.

```
#include<reg51.h>               /*prepocessor directive */

void main()
{

    unsigned int i;             /*local variable*/

    P0=0x00;
    while(1)
     {
     P0=oxff;                   /*statements*/
     for(i=0;i<255;i++);
     P0=0x00;
     for(i=0;i<255;i++);
     }
```

2. write a program to toggle P1.5 of 8051 microcontroller

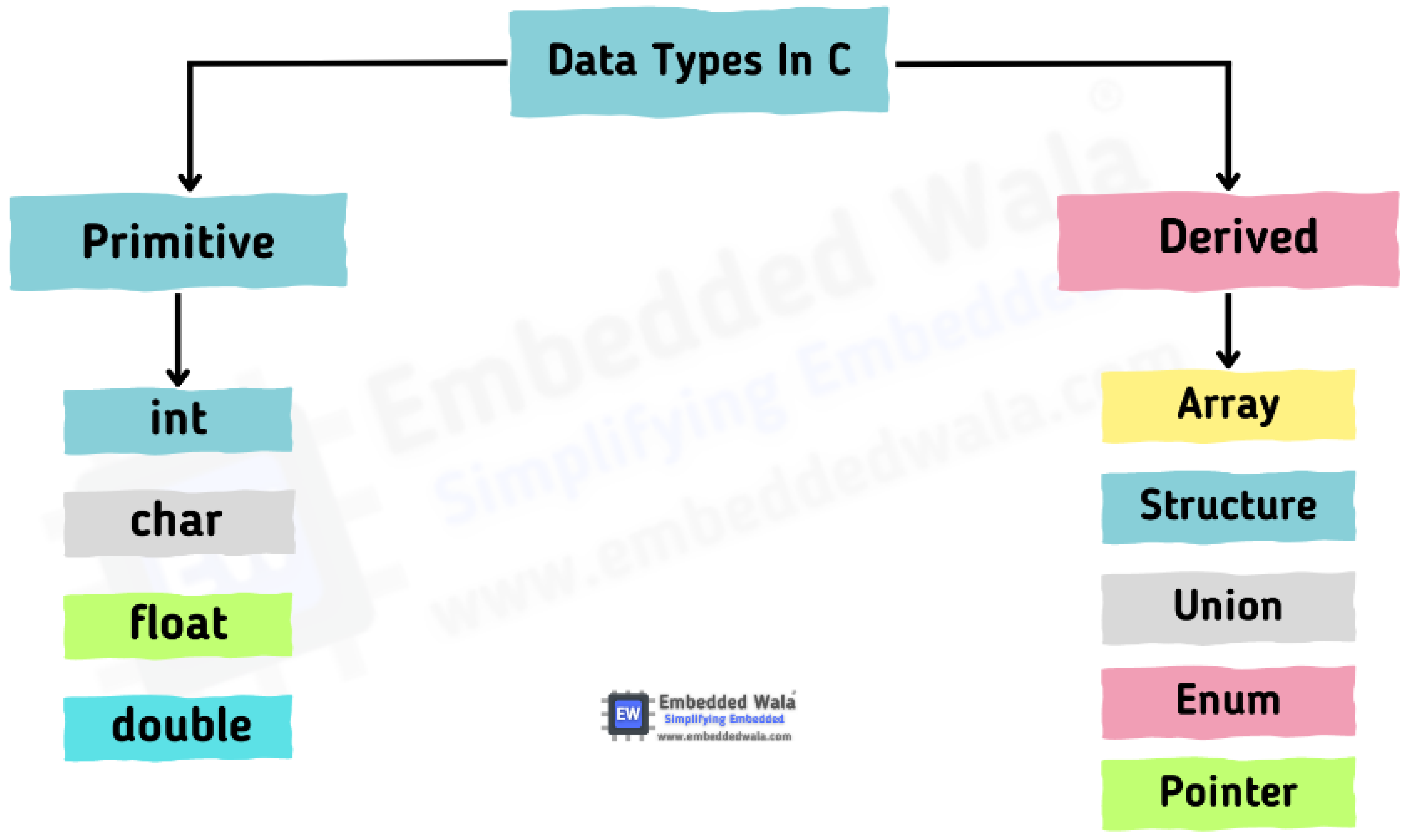
```
#include<reg51.h>

 sbit a=p1^5;

 void main()
 {

      unsigned int k;
      a=0x00;
      while(1)
      {
          a=0xff;
         for(i=0;i<255;i++);
          a=0x00;
         for(i=0;i<255;i++);
       }
 }
```

Data Types



- Understanding the data is important as it impacts the way it is stored or retrieved from memory.
- Data types can affect the memory usage, speed, and accuracy of your program.

Data Types

1) Integer Types :

Integers are used to represent whole numbers without decimal points. In embedded C, several integer types are available, including:

int: The standard signed integer type.

unsigned int: An unsigned integer type that only represents positive numbers.

short int: A signed integer with a smaller range than int.

unsigned short int: An unsigned integer with a smaller range than unsigned int.

long int: A signed integer with an extended range.

unsigned long int: An unsigned integer with an extended range.

Data Types

Here are some commonly used fixed-width integer types, these are present in the #include :

int8_t: A signed **8-bit** integer type.
uint8_t: An unsigned **8-bit** integer type.
int16_t: A signed **16-bit** integer type.
uint16_t: An unsigned **16-bit** integer type.
int32_t: A signed **32-bit** integer type.
uint32_t: An unsigned **32-bit** integer type.
int64_t: A signed **64-bit** integer type.
uint64_t: An unsigned **64-bit** integer type.

Data Types

```
int num = 0; unsigned int num = 1;
short int num = 2;
unsigned short int num = 3;
long int num = 4;
unsigned long int num = 5;
int8_t num = 6;
uint8_t num = 7;
int16_t num = 8;
uint16_t num = 9;
int32_t num = 10;
uint32_t num = 11;
int64_t num = 12;
uint64_t num = 13;
```

Data Types

2) Floating-Point Types :
Floating-point types are used to represent real numbers, including numbers with decimal points. The two commonly used floating-point types in embedded C are:

float: Represents **single-precision** floating-point numbers.
double: Represents **double-precision** floating-point numbers with increased precision compared to float.

float fl = 1.1F;
double dl = 1.2F;

Data Types

3) Bool Type :

A **bool * pointer** is used to store the memory address of a bool variable. By utilizing a **bool * pointer**, we can access and manipulate the value of a bool variable indirectly. **#include** header file is a standard C library header that provides the necessary definitions for using the bool type

bool num = true;

4) Pointer Types :

Pointer types are used to store memory addresses, allowing access to data located at those addresses. Some common pointer types include:

int *: A pointer to an integer.

float *: A pointer to a float.

void *: A pointer to an unspecified type, commonly used for generic pointers.

All types of data can be a pointer type

Data Types

5) Character :

This type of data is used to store single characters like **letters, symbols, or numbers**. The char keyword represents this data type in C.

char ch = 'a';

6) Array:

This type of data is a collection of variables with the same data type and is ideal for storing multiple values of the same type.

int arr[5] = {1, 2, 3, 4, 5};

char ch_arr[5] = {'A', 'B', 'C', 'D', 'E'};

Data Types

7) Structure:

This data type enables you to group variables of different data types into a single unit, making it useful for storing data related to a single entity such as a person or object.

```
struct num_s {
int one;
char two;
float three;
};
struct num_s num;
num.one = 1;
num.two = '2';
num.three = 3.00F;
```

Control Structures

Embedded C/C++ are the **structural programming** languages where program flow executions can be altered with different types of control structure.

This control structure are used to change the flow of program execution to make a decision to continue the flow from start to end.

Control structure is very helpful to make certain decision on data and making a decision after analyzing the particular data or task or conditions to produce final result at the end of complete cycle execution.

Control Structures

The control structures in Embedded C are used to control the flow of execution of a program.
The following are the most commonly used control structures in Embedded C:

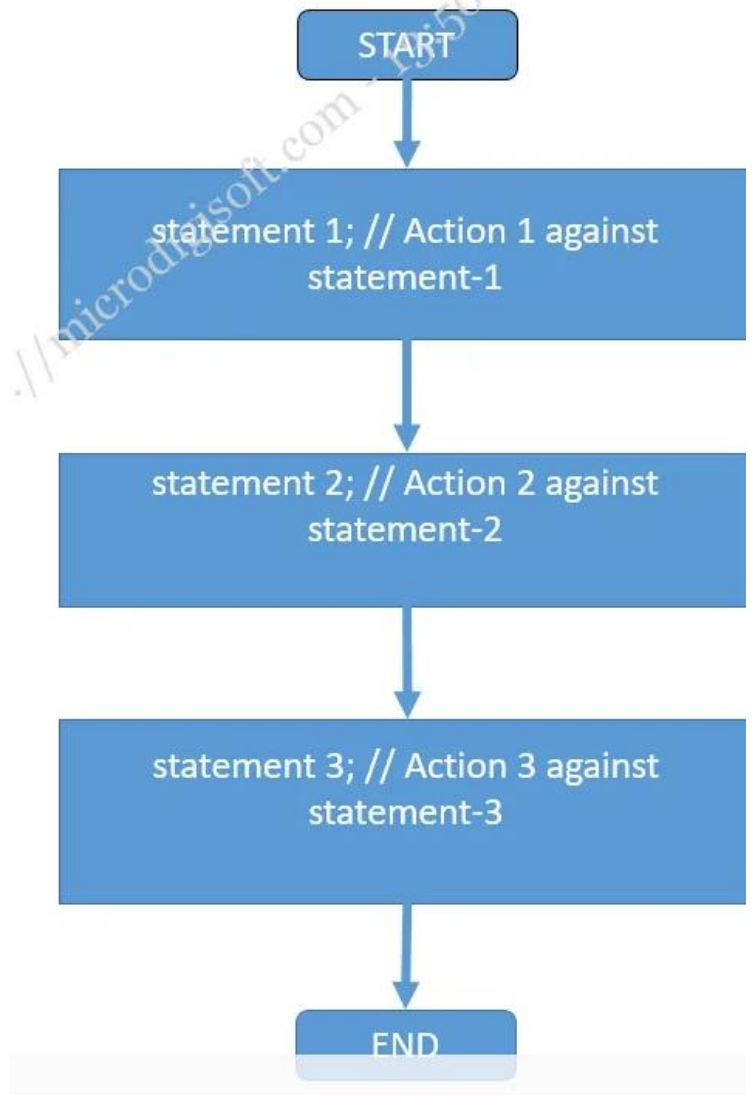
1. **Sequential Control Structure**
2. **Selection Control structure**
3. **Repetition Control Structure**

1. Sequential Control Structure :

Sequential control structure is general and default mode of writing the statements in program. This is case each line of statement written in program code will execute sequentially. It is just like sequential flow diagram.

Control Structures

Sequential Control Structure :


START
statement 1; // Action 1 against statement-1
statement 2; // Action 2 against statement-2
statement 3; // Action 3 against statement-3
END

Control Structures

Selection Control structure:

This type of control structure is used for decisions, branching the flow between 2 or more alternative paths. If and If Else statements are 2 way branching statements where as Switch is a multi branching statement. Conditional statements are used to make decisions based on certain conditions. The most commonly used conditional statements in Embedded C are if-else and switch-case statements.

if

if/else

switch

Control Structures

Selection Control structure:

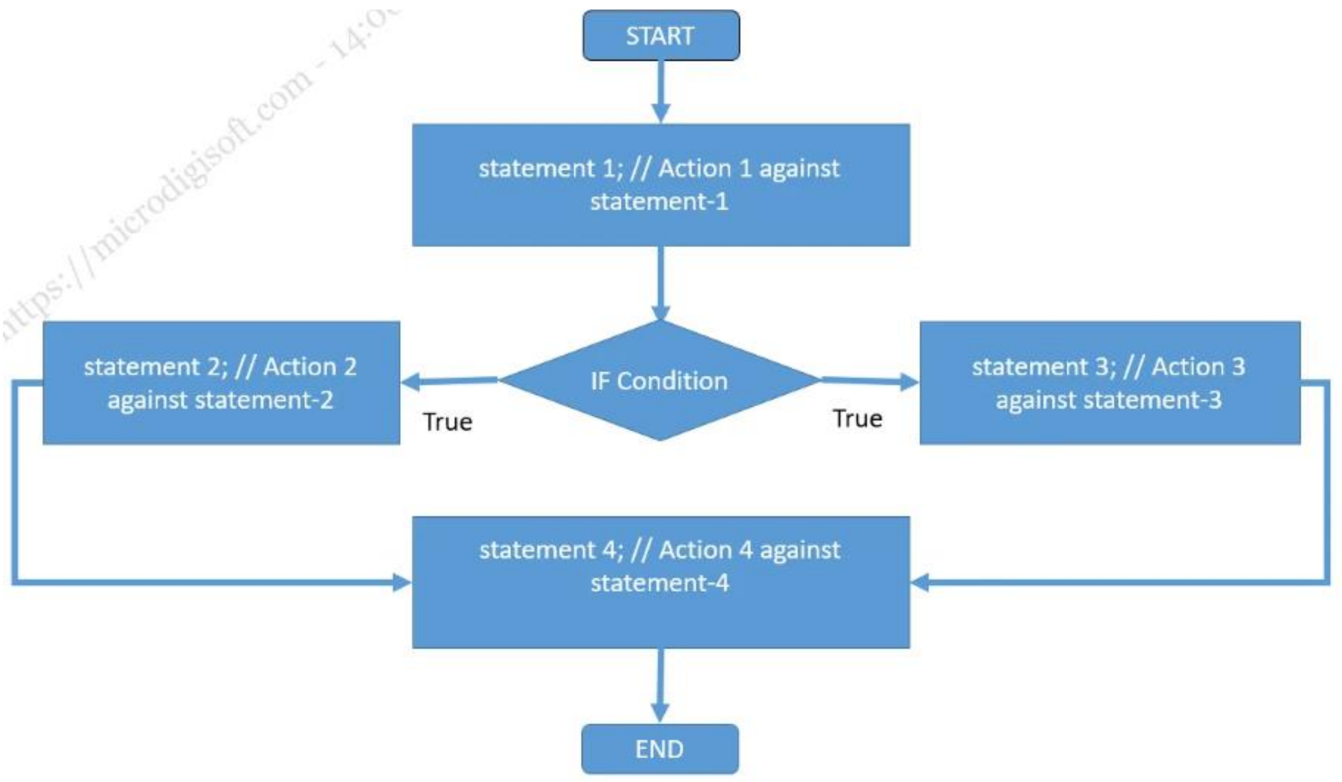
if (expression)   // This expression is evaluated. If expression is TRUE statements execute below statement

{ statement 1;

statement 2; }

 statement 1; // if above expression false Program control is transferred directly to statement 1 and statement 2

statement 2;


START
statement 1; // Action 1 against statement-1
statement 2; // Action 2 against statement-2
True
IF Condition
True
statement 3; // Action 3 against statement-3
statement 4; // Action 4 against statement-4
END

Control Structures

Example Syntax : if /else Statement

if(expression 1) // This Expression 1 is evaluated, if TRUE statement1 & 2 inside the curly braces will be executed.

{ //If FALSE program control is transferred to immediate else if statement.

statement 1;

statement 2; }

else if(expression 2)// If expression 1 is FALSE, expression 2 is evaluated.

{ statement 1;

statement 2; }

else if(expression 3) // If expression 2 is FALSE, expression 3 is evaluated

{ statement 1; statement 2; }

else // If all expressions (1, 2 and 3) are FALSE, the statements that follow this else (inside curly braces) is executed.

{ statement 1; statement 2; } other statements;

Control Structures

Switch statement

The word break is a **keyword in C/C++** used to break from a block of curly braces.

switch(expression) // Expression is evaluated. The outcome of the expression should be an integer or a character constant

{

case value1: // case is the keyword used to match the integer/character constant from expression.

//value1, value2 ... are different possible values that can come in expression

statement 1;

statement 2;

break; // break is a keyword used to break the program control from

}

Control Structures

switch block.
case value2:
statement 1;
statement 2;
break;
default: // default is a keyword used to execute a set of statements inside switch, if no case values match the expression value.
statement 1;
statement 2;
break;
}

Control Structures

Repetition Control Structure

As the name suggest the repetition control structure is used to execute repetitive task inside the loop.

The repetition control structure in C language means repeating a piece of code multiple times in a row.

Looping statements are used to repeat a block of code multiple times. The most commonly used looping statements in Embedded C are

for, while, and do-while loops.

```
for(initialization statements; test condition; iteration statements)
 { statement 1; statement 2; statement 3; }
```

Control Structures

Do while

```
Do
 { statement 1;
statement 2;
statement 3; }
while(condition);
```

While

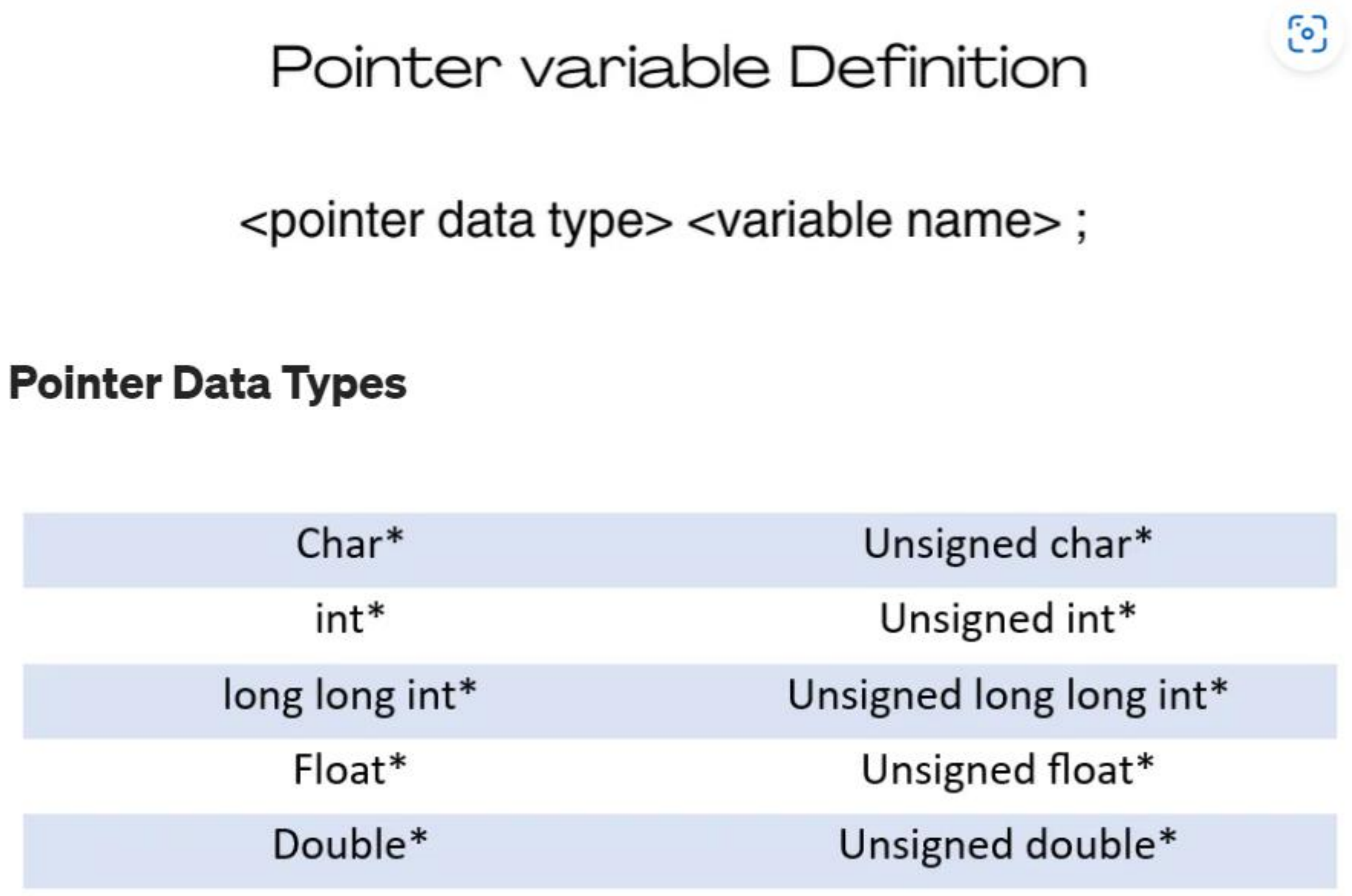
```
while(condition)// This condition is tested for TRUE or FALSE.
Statements inside curly braces are executed as long as condition is TRUE
{ statement 1;
statement 2;
 }
```

Pointers

- Pointers play a crucial role in memory management, serving as variables that hold the address of another variable and enabling the manipulation of memory addresses.
- They offer direct access to memory, facilitating more efficient data handling.
- Additionally, pointers enable the return of multiple values from functions, contribute to memory space conservation, and enhance execution speed by directly accessing memory locations during data manipulation.

- They can reference the same memory space from different locations and assist in dynamic memory allocation and deallocation, highlighting their importance.

Pointers Variable



Pointer variable Definition

<pointer data type> <variable name> ;

Pointer Data Types

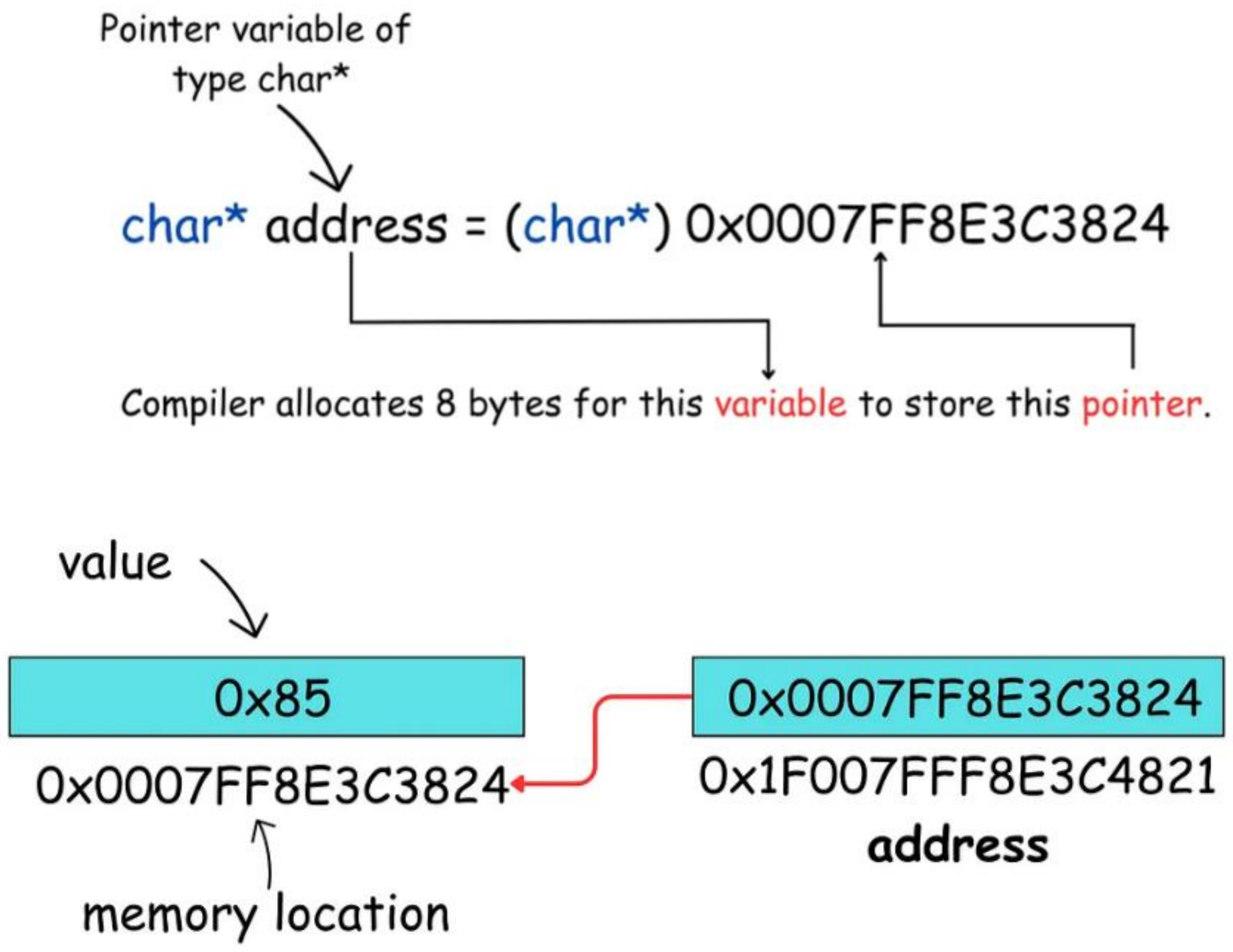
Char*	Unsigned char*
int*	Unsigned int*
long long int*	Unsigned long long int*
Float*	Unsigned float*
Double*	Unsigned double*

Pointers Initalization

-
  ```
  int a = 10;
  int *ptr; //pointer declaration
  ptr = &a; //pointer initialization with int variable a
  ```

- Once a pointer is declared, it’s crucial to initialize it before using it in a program.
- Pointer initialization involves assigning the address of a variable to the pointer variable.
- In the ‘C’ language, the address operator “&” is employed to find the address of a variable.

Pointers Initalization


Pointer variable of type char*
char* address = (char*) 0x0007FF8E3C3824
Compiler allocates 8 bytes for this variable to store this pointer.
value
0x85
0x0007FF8E3C3824
memory location
0x0007FF8E3C3824
0x1F007FFF8E3C4821
address

Pointers

```
//read operation on address variable yields 1 byte of data
char* address = (char*) 0x0007FF8E3C3824

//read operation on address variable yields 4 byte of data
int* address = (int*) 0x0007FF8E3C3824

//read operation on address variable yields 8 byte of data
long long int* address = (long long int*) 0x0007FF8E3C3824
```

Pointers

How is Pointer used in embedded programming?

- Pointers allow **direct access to memory locations**, facilitating efficient manipulation of data. In embedded systems with limited resources, direct memory manipulation can be more efficient than using higher-level abstractions.
- Pointers are essential for managing dynamically allocated memory, allowing for efficient use of resources.
- Pointers can be used to access and manipulate registers, allowing interaction with various peripherals such as GPIO
- Pointers are used to work with structures and dynamic data structures like linked lists. This allows for the creation of flexible and extensible data structures, even in resource-constrained environments.

Memory Management

1. **Static Memory Allocation**:

Static memory allocation involves allocating memory at compile time. Variables declared globally or with the “static” keyword are allocated statically and remain in memory throughout the program’s execution. While static allocation is straightforward and efficient in terms of memory usage, it lacks flexibility as the memory allocation cannot be changed dynamically during runtime.

```
static int staticVariable;  // Static variable
```

2. Dynamic Memory Allocation:

Dynamic memory allocation allows for flexible memory allocation during runtime using functions like malloc(), calloc(), realloc(), and free().

Memory Management

Dynamic Memory Allocation (Contd..):
However, dynamic memory allocation comes with overhead and the risk of memory fragmentation, making it less suitable for resource-constrained embedded systems.
int *dynamicArray = malloc(10 * sizeof(int)); // Dynamic memory allocation
Use **free ()** to clear up memory once not needed.

3. Memory Pools: Memory pools are pre-allocated blocks of memory divided into fixed-size chunks. Memory pools provide a compromise between static and dynamic memory allocation, offering flexibility while minimizing fragmentation and overhead.

Memory Management

Memory Pools (Contd..)

They are particularly useful in real-time embedded systems where deterministic memory allocation is required.

```
#define POOL_SIZE 100
char memoryPool[POOL_SIZE];
```

Optimization Techniques:

Optimizing embedded C code is essential for improving performance, reducing memory usage, and extending battery life in battery-powered devices.

Here are some optimization techniques commonly employed in embedded software development:

Memory Management

Code Optimization:

Optimizing code involves writing efficient algorithms and minimizing unnecessary operations. Techniques such as loop unrolling, function inlining, and reducing branching can significantly improve code performance.

```
// Example: Loop Unrolling
for (int i = 0; i < 10; i += 2) {
// Loop body
// Code for iteration i
// Code for iteration i+1
}
```

Memory Management

Memory Optimization:

- Reducing memory usage is critical in embedded systems with limited RAM and ROM.
- Techniques such as using smaller data types, optimizing data structures, and removing unused code can help conserve memory.

// Example: Using Smaller Data Types

uint8_t value = 10; // Use uint8_t instead of int if the range of values is within 0-255

Compiler Optimization:

Modern compilers offer various optimization options to improve code performance and size.

Memory Management

Compiler Optimization (Contd..):

Enabling optimization flags (-O1, -O2, -Os) during compilation can instruct the compiler to apply optimizations such as dead code elimination, function inlining, and loop optimization.

```
gcc -O2 -o output_file input_file.c
```

AVR Interrupt Handling

- The AVR's global Interrupts Enable bit must be set to one in the microcontroller control register SREG.
- This allows the AVR's core to process interrupts via ISRs when set, and prevents them from running when set to zero.
- It is like a global ON/OFF switch for the interrupts. By default this bit is zero.

There are two main sources of interrupts:

Hardware Interrupts : which occur in response to a changing external event such as a pin going low, or a timer reaching a preset value

Software Interrupts : which occur in response to a command issued in software

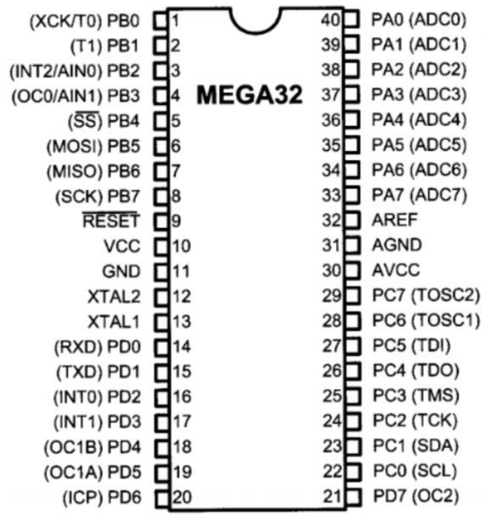
AVR Interrupt Handling

- In 8-bit AVRs the software interrupts are not available, which are basically used for generating user defined Exceptions and handling them as and when they occur.
  Each microcontroller has a set of interrupt sources available.
  Some of the interrupts available in ATmega16 are as follows:
- External Interrupt
- Timer Interrupt
- USART Receive and Transmit Interrupt
- EEPROM Ready Interrupt
- ADC conversion complete

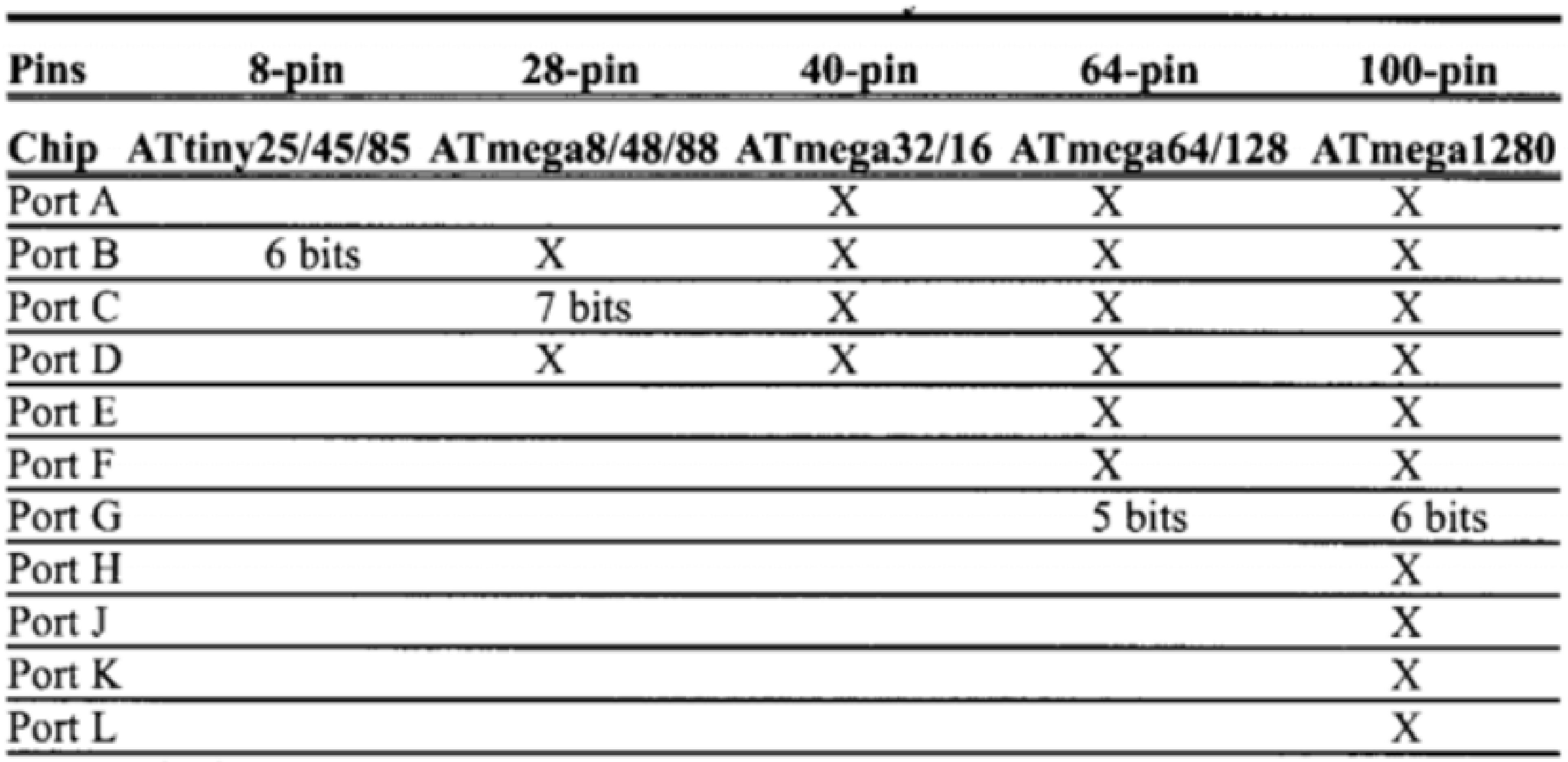
Input and Output Ports Interfacing on AVR

- In AVR microcontroller family, there are many ports available for I/O operations, depending on which family microcontroller you choose.
- For the ATmega32 40-pin chip 32 Pins are available for I/O operation. The four ports PORTA, PORTB, PORTC, and PORTD are programmed for performing desired operation.
- The Pin diagram of ATmega32 microcontroller is shown below:

The number of ports in AVR family varies depending on number of pins available on chip. The 8-pin AVR has port B only, while the 64-pin version has ports A to ports F, and the 100-pin AVR has ports A to ports L.



Input and Output Ports Interfacing on AVR



Pins	8-pin	28-pin	40-pin	64-pin	100-pin
Chip	ATtiny25/45/85	ATmega8/48/88	ATmega32/16	ATmega64/128	ATmega1280
Port A			X	X	X
Port B	6 bits	X	X	X	X
Port C		7 bits	X	X	X
Port D		X	X	X	X
Port E				X	X
Port F				X	X
Port G				5 bits	6 bits
Port H					X
Port J					X
Port K					X
Port L					X

Note: X indicates that the port is available.

Input and Output Ports Interfacing on AVR

Generally, we use a timer/counter to generate time delays, waveforms, or to count events. Also, the timer is used for PWM generation, capturing events, etc.

In AVR ATmega16 / ATmega32, there are three timers:

Timer0: 8-bit timer

Timer1: 16-bit timer

Timer2: 8-bit timer

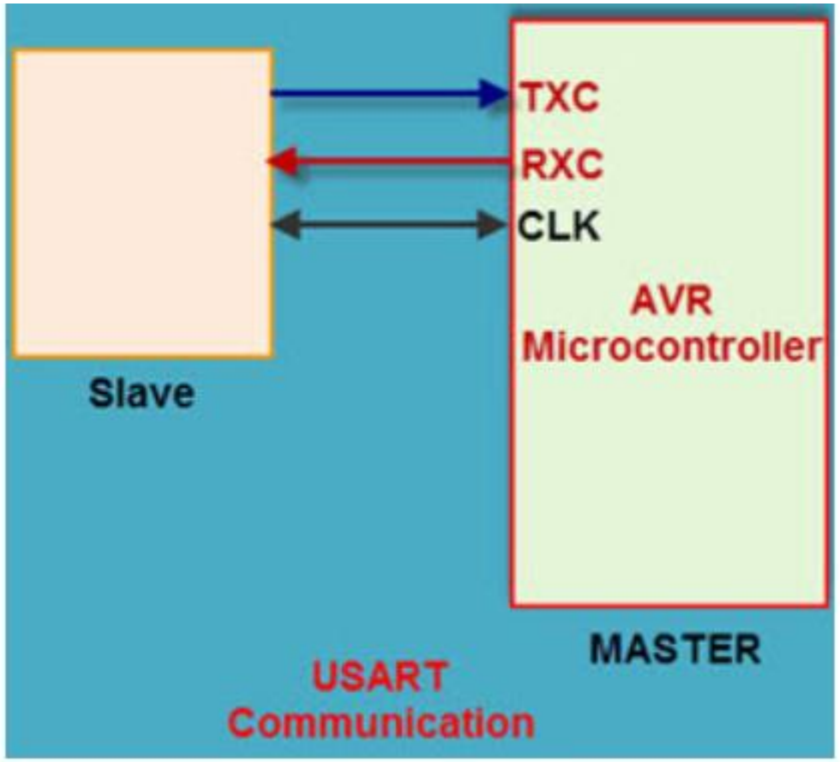
Basic registers and flags of the Timers

Serial Communication in AVR

- The AVR microcontroller has two pins: TXD and RXD, which are specially used for transmitting and receiving the data serially.
- Any AVR microcontroller consists of USART protocol with its own features.

The Main Features of AVR USART

- The USART protocol supports the full-duplex protocol.
- It generates high resolution baud rate.
- It supports transmitting serial data bits from 5 to 9 and it consists of two stop bits.



USART Communication in AVR Microcontroller

Serial Communication in AVR

USART Pin Configuration

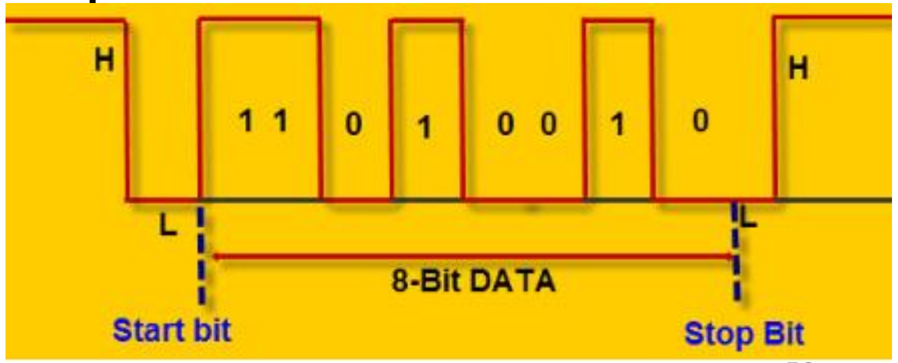
The USART of AVR consists of three Pins:

- RXD: USART receiver pin (ATMega8 PIN 2; ATMega16/32 Pin 14)
- TXD: USART transmitter pin (ATMega8 PIN 3; ATMega16/32 Pin 15)
- XCK: USART clock pin (ATMega8 PIN 6; ATMega16/32 Pin 1)

Modes of Operation

The AVR microcontroller of USART protocol operates in three modes which are:

- Asynchronous Normal Mode
- Asynchronous Double Speed Mode
- Synchronous Mode



Serial Communication in AVR

Asynchronous Normal Mode:

- In this mode of communication, the data is transmitted and received bit by bit without clock pulses by the predefined baud rate set by the UBBR register.

Asynchronous Double Speed Mode

- In this mode of communication, the data transferred at **double the baud rate** is set by the UBBR register and set U2X bits in the UCSRA register. This is a high-speed mode for synchronous communication for transmitting and receiving the data quickly. This system is used where accurate baud rate settings and system clock are required.

Synchronous Mode

- In this system, transmitting and receiving the data with respect to clock pulse is set UMSEL=1 in the UCSRC register.