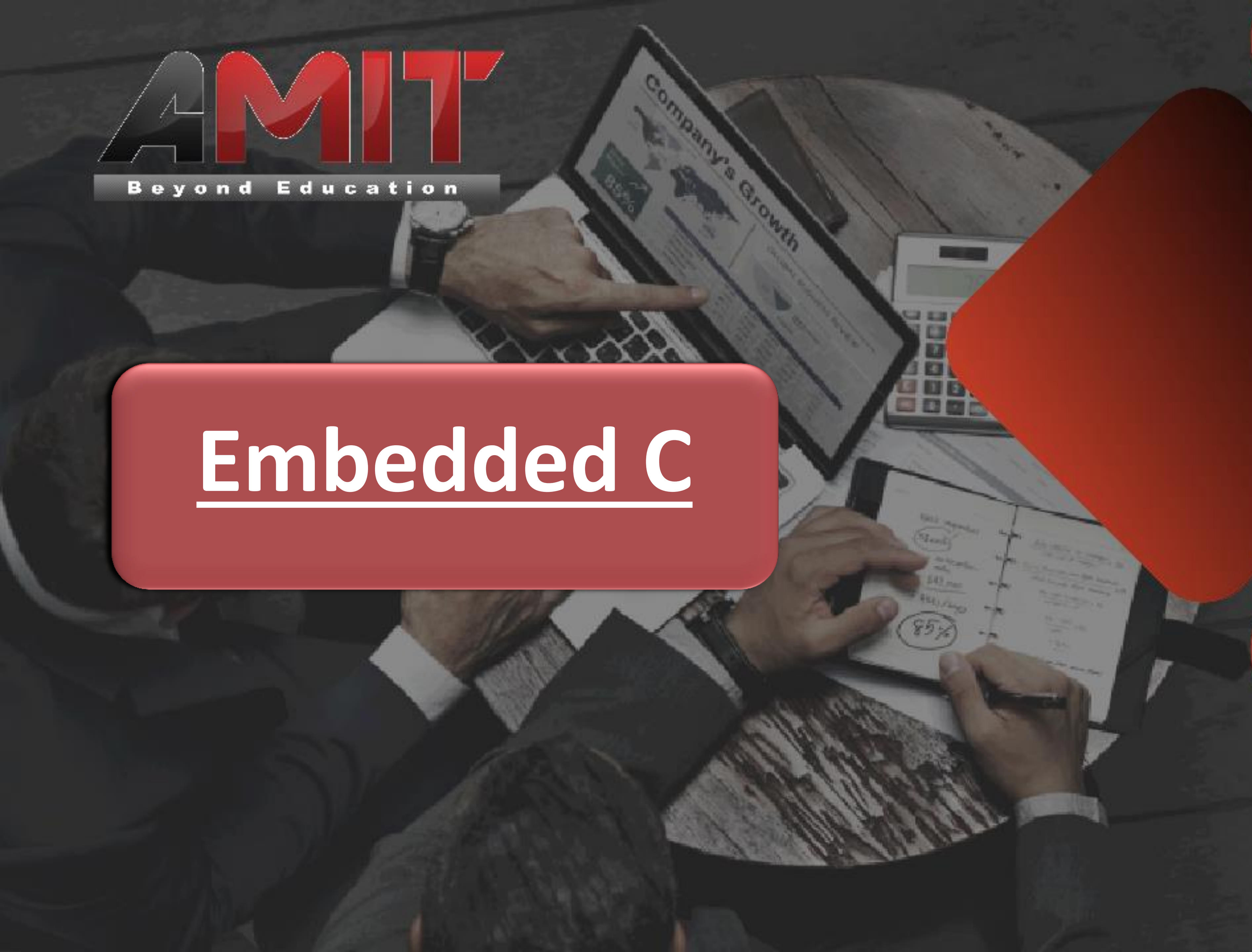
Beyond Education

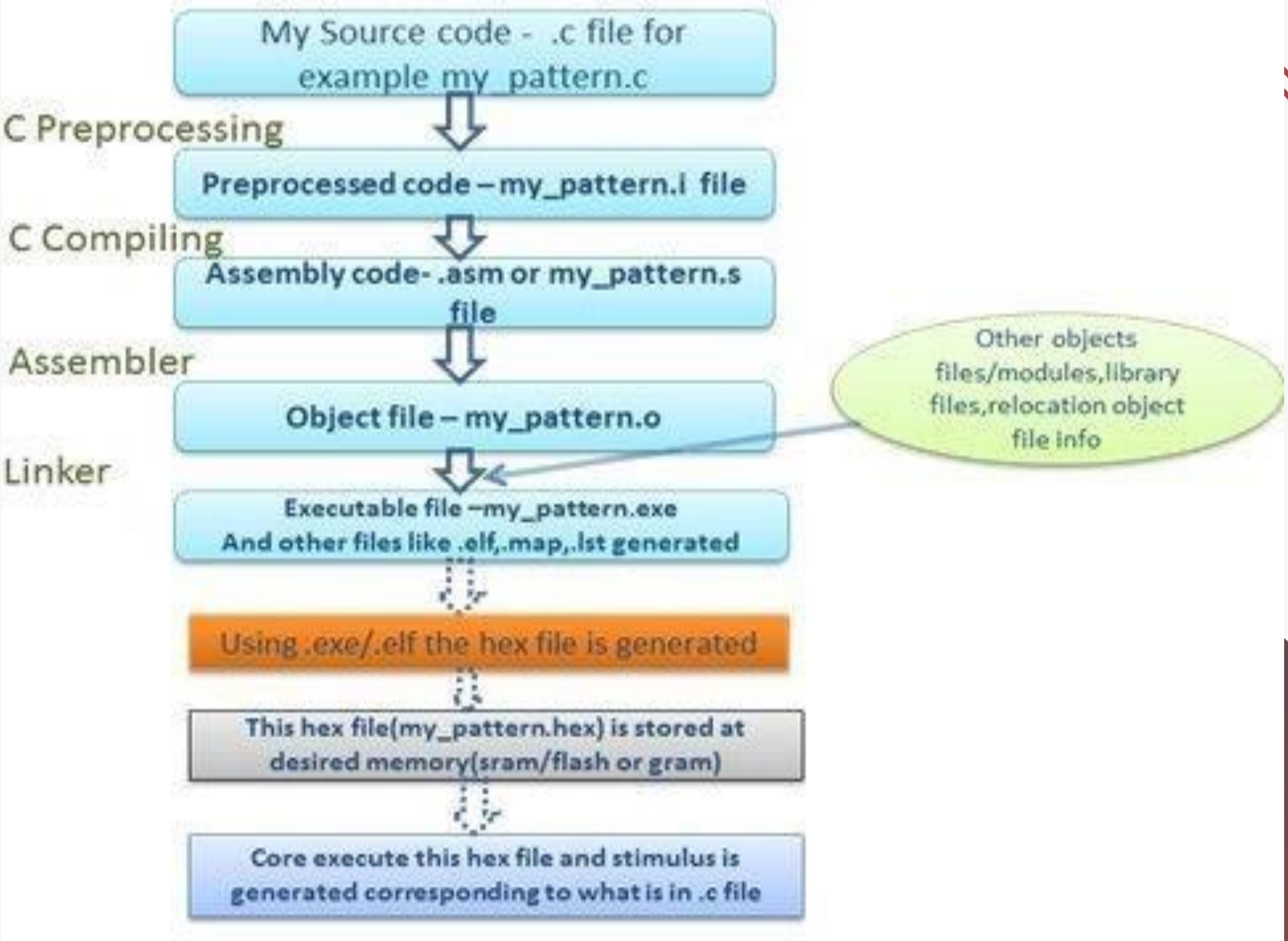
Embedded C



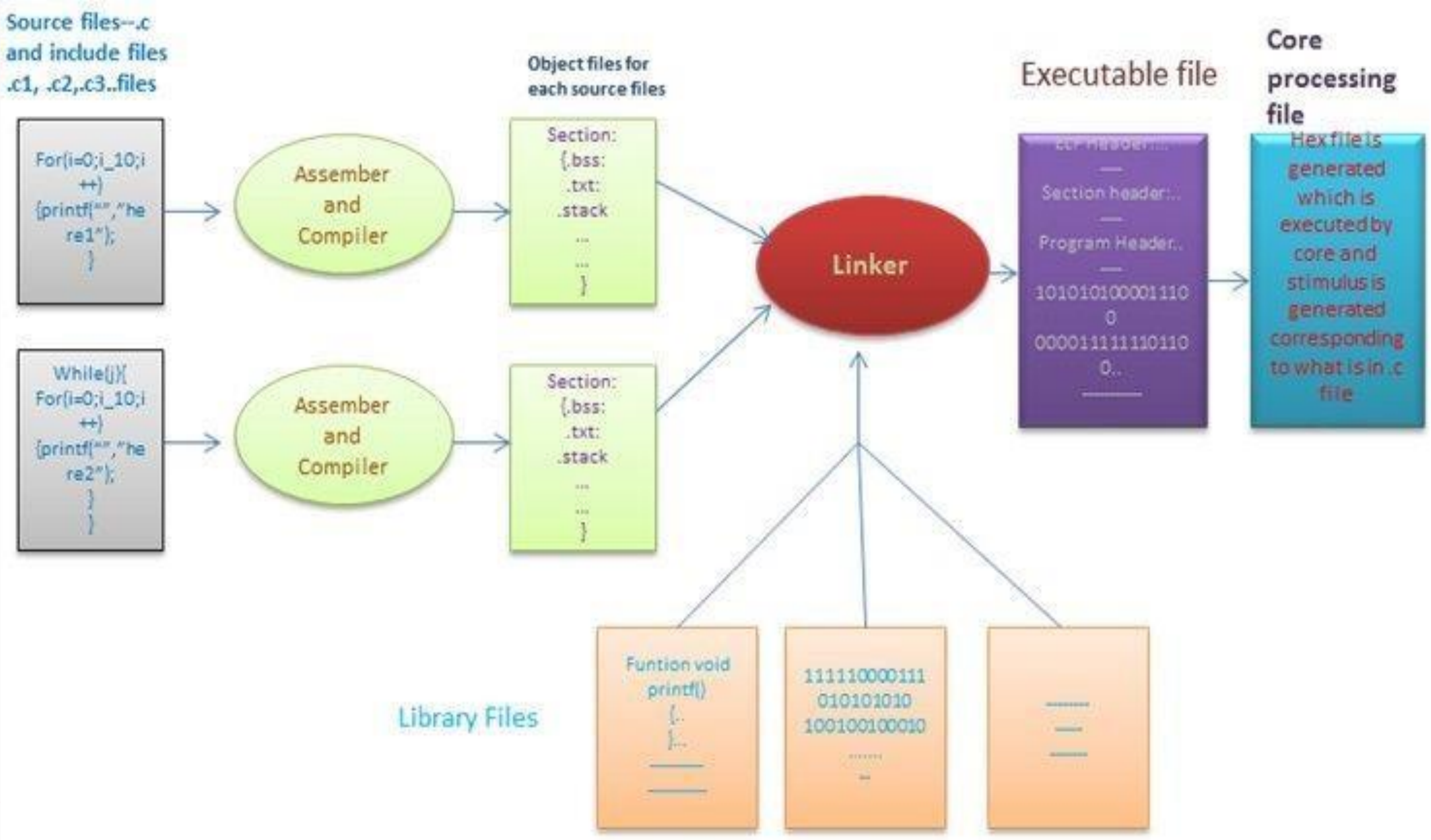
Contents

AMIT Beyond Education

Build Process



Build Process more details



C preprocessor directives

- **Macro substitution directives.** example: #define

- **File inclusion directives.** example: #include

- **Conditional compilation directive.** example: #if, #else, #ifdef, #undef

- **Miscellaneous directive.** example: #error, #line

Preprocessor	Syntax	Description
Macro	#define	This macro defines constant value and can be any of the basic data types.
Header file inclusion	#include <file_name>	The source code of the file "file_name" is included in the main program at the specified place
Conditional compilation	#ifdef, #endif, #if, #else, #ifndef	Set of commands are included or excluded in source program before compilation with respect to the condition
Other directives	#undef, #pragma	#undef is used to undefine a defined macro variable. #Pragma is used to call a function before and after main function in a C program

AMIT Beyond Education

C Preprocessor

We'll refer to the C Preprocessor as CPP.

Preprocessors Examples

Analyze the following examples to understand various directives.

#define MAX_ARRAY_LENGTH 20
This directive tells the CPP to replace instances of MAX_ARRAY_LENGTH with 20. Use #define for constants to increase readability.

#include <stdio.h> #include "myheader.h"
These directives tell the CPP to get stdio.h from System Libraries and add the text to the current source file. The next line tells CPP to get myheader.h from the local directory and add the content to the current source file.

#undef FILE_SIZE #define FILE_SIZE 42
It tells the CPP to undefine existing FILE_SIZE and define it as 42.

Sr.No.	Directive & Description
1	**#define** Substitutes a preprocessor macro.
2	**#include** Inserts a particular header from another file.
3	**#undef** Undefines a preprocessor macro.
4	**#ifdef** Returns true if this macro is defined.
5	**#ifndef** Returns true if this macro is not defined.
6	**#if** Tests if a compile time condition is true.
7	**#else** The alternative for #if.

C Preprocessor

- #ifndef MESSAGE
-   #define MESSAGE "You wish!"
- #endif
- It tells the CPP to define MESSAGE
- only if MESSAGE isn't already defined.

- #ifdef DEBUG
  /* Your debugging statements here */
- #endif

AMIT
Beyond Education

8	**#elif** #else and #if in one statement.
9	**#endif** Ends preprocessor conditional.
10	**#error** Prints error message on stderr.
11	**#pragma** Issues special commands to the compiler, using a standardized method.

AMIT
Beyond Education

Object Files

- The important fields of object file are :
- **.text** :This section contains the executable instruction codes and is shared among every process running the same binary. This section usually has READ and EXECUTE permissions only. This section is the one most affected by optimization.

- **.bss**: BSS stands for ‘Block Started by Symbol’. It holds un-initialized global and static variables. Since the BSS only holds variables that don't have any values yet, it doesn't actually need to store the image of these variables. The size that BSS will require at runtime is recorded in the object file, but the BSS (unlike the data section) doesn't take up any actual space in the object file.

Object file contents

- **.data**: Contains the initialized global and static variables and their values. It is usually the largest part of the executable. It usually has READ/WRITE permissions.
- **.reloc**: Stores the information required for relocating the image while loading.

Bit Math

<table>
<tr><th>SHIFT LEFT</th><th>SHIFT RIGHT</th><th>NOT</th><th>AND</th><th>OR</th><th>XOR</th></tr>
<tr><td><<</td><td>>></td><td>~</td><td>&</td><td>|</td><td>^</td></tr>
<tr><td>1 << 2
=
0000 0100
or
4</td><td>8 >> 2
=
0000 0010
or
2</td><td>~0 = 1
~1 = 0</td><td>0 & 0 = 0
0 & 1 = 0
1 & 0 = 0
1 & 1 = 1</td><td>0 | 0 = 0
0 | 1 = 1
1 | 0 = 1
1 | 1 = 1</td><td>0 ^ 0 = 0
0 ^ 1 = 1
1 ^ 0 = 1
1 ^ 1 = 0</td></tr>
</table>

Commonly used Bitwise Operations

- Operate on Bit Vectors
  - Operations applied bitwise

```
  01101001      01101001      01101001
& 01010101    | 01010101    ^ 01010101    ~ 01010101
  --------      --------      --------      --------
  01000001      01111101      00111100      10101010
```

- All of the Properties of Boolean Algebra Apply

AMIT
Beyond Education

MASKING

- Bit Masking is used to get value of a specific bit
- It’s used as following:

AMIT
Beyond Education

```
x = 64;
y = x & 0b00101000;    // y = 0 if 5th or 7th bits are not true, and y>0 if one or both are  true
```

Mathematically here is what we did above:

```
  0100 0000          x (set to 64 on the first line)
& 0010 1000          mask (created with 0b00101000 on the second line)
  ---------
  0000 0000          result, loaded into y
```

Bit Masking (Example)

```
x = 64;
y = x & ( (1<<5) | (1<<3) );
```

Mathematically here is what we did:

Solve the brackets:

(1 << 3)
creates 0000 0001
shift it left by 3 to get 0000 1000

(1 << 5)
create 0000 0001
shift it left by 5 to get 0010 0000

Rearranged to solve the ((1<<5) | (1<<3))
part of the equation:

```
    0000 1000   (1 << 3)
  | 0010 0000   (1 << 5)
-----------------
  0010 1000     notice that we just created 0b00101000
```

Substitute:
y = x & 0010 1000
Now rearrange to solve:

```
    0100 0000
  & 0010 1000        x (set to 64 on the first line)
  (1<<3)) )          mask (created with ( ((1<<5) |

  0000 0000     result, loaded into y
-------------
```

PUTTING IT ALL TOGETHER:

```
y = 64;
y |= (1<<3);
```

Expand:
y = y | (1 << 3)
Solve the brackets:

(1 << 3)
creates 0000 0001
shift it left by 3 to get a 0000 1000

Substitute:
y = y | 0000 1000

And finally rearrange to solve:

```
  0100 0000   y (set to 64 on the first line)
| 0000 1000   mask (created with (1<<3) )
  ----------
  0100 1000     result, loaded into y
```

LAB BITWISE

- Create a header file that provide the equations of bit wise operations as macros
- For example:

- SET_BIT(REG,BIT) REG|=(1<<BIT)
- GET_BIT(REG,BIT) ((REG>>BIT) & 1)
- CLR_BIT(REG,BIT) REG&=~(1<<BIT)
- TOGGLE_BIT(REG,BIT) REG^=(1<<BIT)

Constant and Volatile Qualifiers

- **const** is used with a data type declaration or definition to specify an unchanging value and its placed in .rodata section in ROM
- Examples:

```
const int five = 5;
const double pi = 3.141593;
```

const objects may not be changed

- The following are illegal:

```
const int five = 5;
const double pi = 3.141593;

pi = 3.2;
five = 6;
```

AMIT Beyond Education

Volatile Qualifier

- volatile specifies a variable whose value may be changed by processes outside the current program
- One example of a volatile object might be a buffer used to exchange data with an external device:

```
volatile int iobuf;
Int check_iobuf(void)
{
        int val;

        while (iobuf == 0) {
        }
        val = iobuf;
        iobuf = 0;
        return(val); }
```

- if iobuf had not been declared volatile, the compiler would notice that nothing happens inside the loop and thus eliminate the loop
- const and volatile can be used together

- An input-only buffer for an external device could be declared as const volatile (or volatile const, order is not important) to make

  sure the compiler knows that the variable should not be changed (because it is input-only) and that its value may be altered by processes other than the current program

When to use volatile?

Interview Question

Registers that are modified by hardware

- Global variable used inside interrupt

Resources inside task will be needed by another task wont be spawned yet

Problems that makes you use volatile

- **Code that works fine--until you enable compiler optimizations**
- **Code that works fine--until interrupts are enabled**
- **Flaky hardware drivers**
- **RTOS tasks that work fine in isolation--until some other task is spawned**

AMIT Beyond Education

Optimization

GCC Compiler provides an option to optimize the code to reflect :

1) Execution time 2) Code size
3)Memory Usage
4) Compilation times

There are levels of Optimization provided by
option flag

gcc -o option flag

Write the build output to an output file.

Syntax

```
$ gcc [options] [source files] [object files] -o output file
```

Example

myfile.c:

```
// myfile.c
#include <stdio.h>

void main()
{
    printf("Program run\n");
}
```

Build *myfile.c* on terminal and run the output file *myfile*:

```
$ gcc myfile.c -o myfile
$ ./myfile
Program run
$
```

AMIT Beyond Education

Optimization

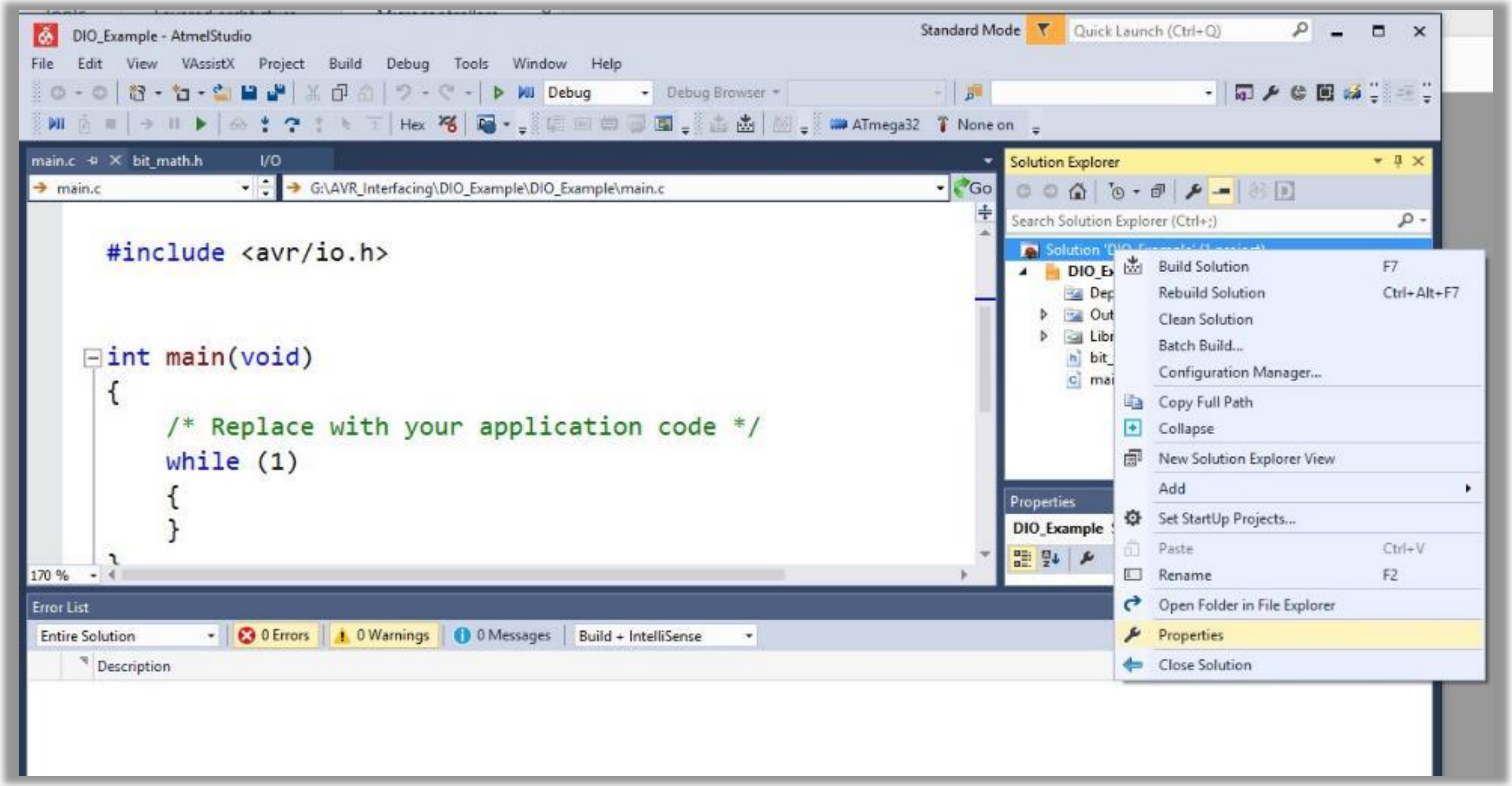
gcc -O option flag

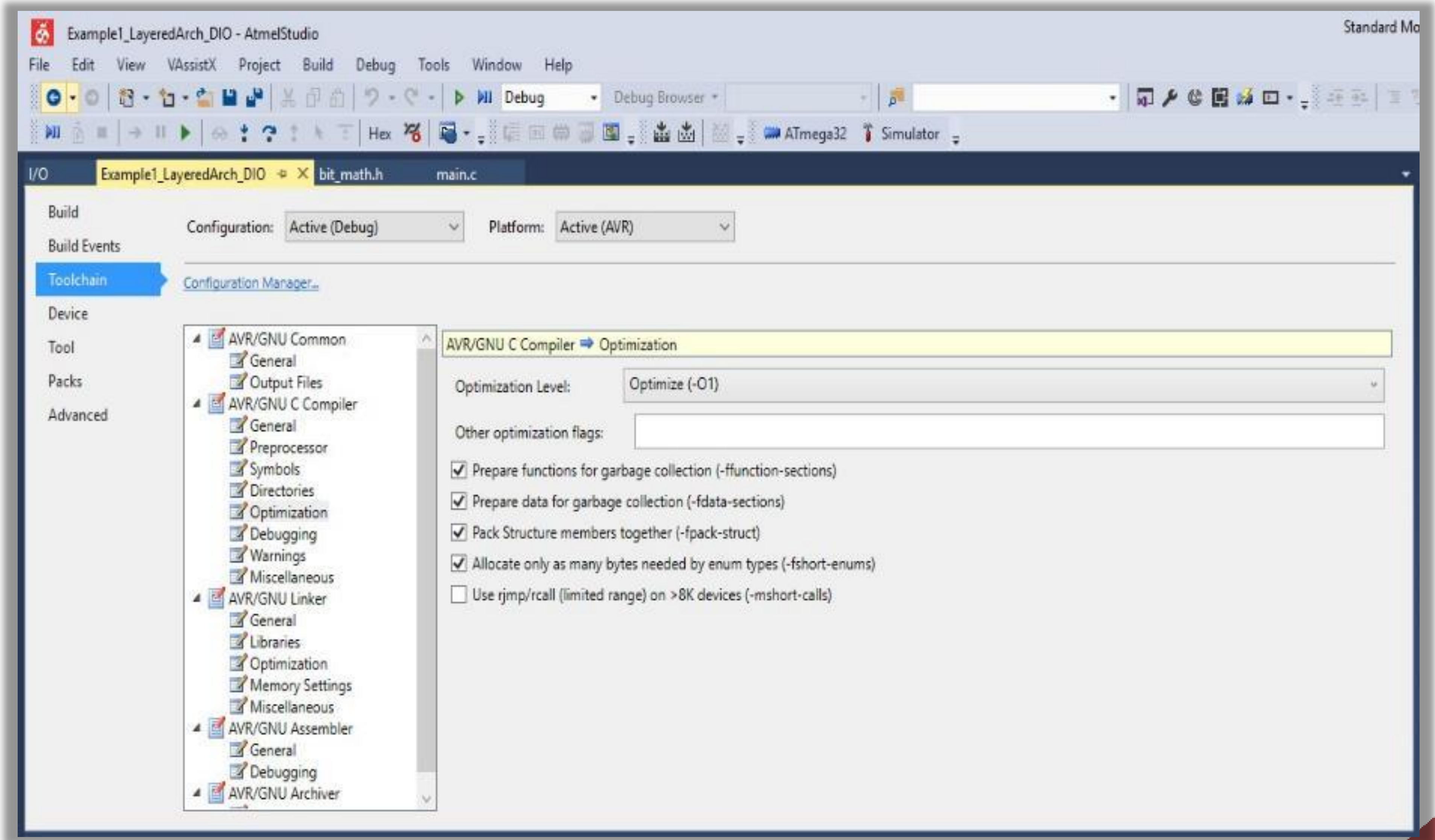
Set the compiler's optimization level.

option	optimization level	execution time	code size	memory usage	compile time
-O0	optimization for compilation time (default)	+	+	-	-
-O1 or -O	optimization for code size and execution time	-	-	+	+
-O2	optimization more for code size and execution time	--		+	++
-O3	optimization more for code size and execution time	---		+	+++
-Os	optimization for code size		--		++
-Ofast	O3 with fast none accurate math calculations	---		+	+++

+increase ++increase more +++increase even more -reduce --reduce more ---reduce even more

AMIT
Beyond Education


DIO_Example - AtmelStudio
File Edit View VAssistX Project Build Debug Tools Window Help
main.c bit_math.h I/O
G:\AVR_Interfacing\DIO_Example\DIO_Example\main.c
#include <avr/io.h>
int main(void)
{
/* Replace with your application code */
while (1)
{
}
Solution Explorer
Build Solution F7
Rebuild Solution Ctrl+Alt+F7
Clean Solution
Batch Build...
Configuration Manager...
Copy Full Path
Collapse
New Solution Explorer View
Add
Set StartUp Projects...
Paste Ctrl+V
Rename F2
Open Folder in File Explorer
Properties
Close Solution
Error List
Entire Solution
0 Errors
0 Warnings
0 Messages
Build + IntelliSense
Description


Example1_LayeredArch_DIO - AtmelStudio
File Edit View VAssistX Project Build Debug Tools Window Help
Debug
Debug Browser
Hex
ATmega32
Simulator
I/O
Example1_LayeredArch_DIO
bit_math.h
main.c
Build
Build Events
Toolchain
Device
Tool
Packs
Advanced
Configuration: Active (Debug)
Platform: Active (AVR)
Configuration Manager...
AVR/GNU Common
General
Output Files
AVR/GNU C Compiler
General
Preprocessor
Symbols
Directories
Optimization
Debugging
Warnings
Miscellaneous
AVR/GNU Linker
General
Libraries
Optimization
Memory Settings
Miscellaneous
AVR/GNU Assembler
General
Debugging
AVR/GNU Archiver
AVR/GNU C Compiler ⇒ Optimization
Optimization Level: Optimize (-O1)
Other optimization flags:
Prepare functions for garbage collection (-ffunction-sections)
Prepare data for garbage collection (-fdata-sections)
Pack Structure members together (-fpack-struct)
Allocate only as many bytes needed by enum types (-fshort-enums)
Use rjmp/rcall (limited range) on >8K devices (-mshort-calls)

Enum Cont’

- The GCC C compiler will allocate enough memory for an enum to hold any of the values that you have declared. So, if your code only uses values below 256, your enum should be 8 bits wide.

- If you have even one value that is greater than 255, C will make the enum larger than 8 bits; big enough to hold the biggest number in the enum.

```
typedef enum
{
        Dio_Port_A,
        Dio_Port_B,
        Dio_Port_C,
        Dio_Port_D
}Dio_PortType;
```

AMIT Beyond Education

Enum

- Enums are a great way to put descriptive names on "magic numbers", unexplained values that litter code and really should be avoided.

- Int value=5; //5 is a magic number
- The C standard specifies that enums are integers, but it does not specify the size. Once again, that is up to the people who write the compiler. On an 8-bit processor, enums can be 16-bits wide. On a 32-bit processor they can be 32-bits wide or more or less.

typedef

typedef is a keyword used in C language to assign alternative names to existing datatypes. Its mostly used with user defined datatypes, when names of the datatypes become slightly complicated to use in programs.

typedef unsigned char uint8; typedef unsigned short int uint16; typedef unsigned long int uint32;

typedef can be used to give a name to user defined data type as well. Lets see its use with structures.

```
typedef struct { type member1;
type member2;
type member3;
 } type_name;
```

- Typedef Vs Macro?

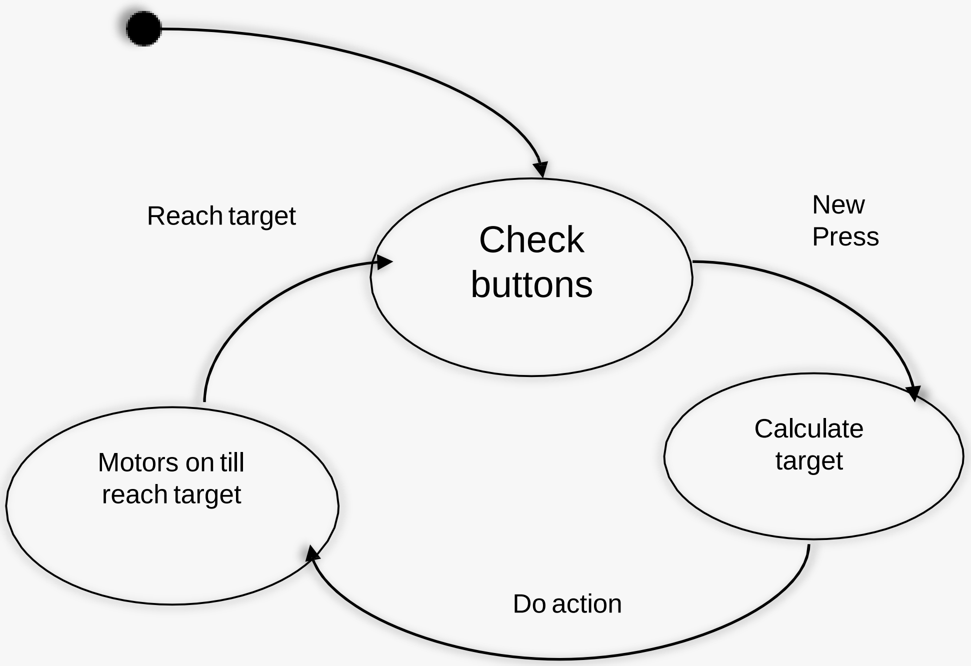
I.V Question

LAB 2

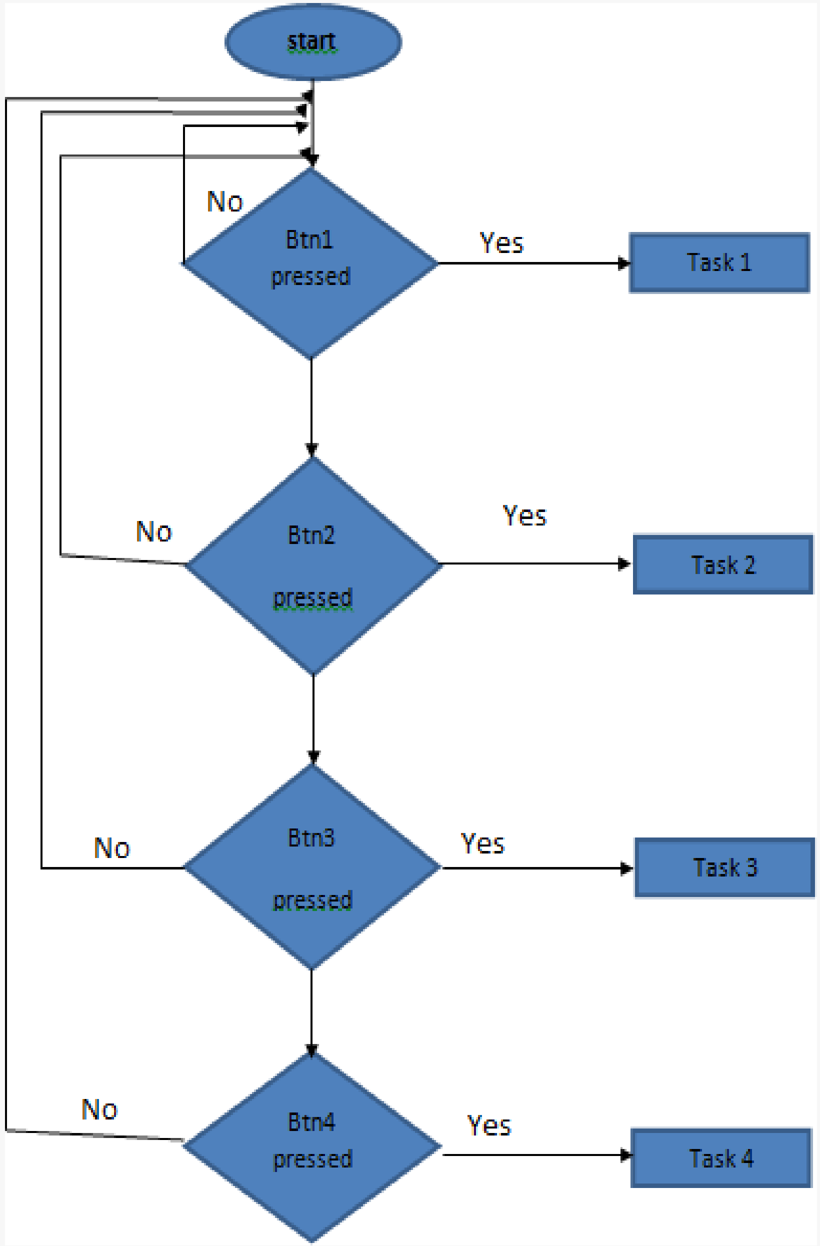
- Create a Standard type definitions (STD_Type.h)
- uint8 , uint16 and uint32 and other for the signed integer
- Also place port and pin standard types using enum

Design Concepts

- There are two types of design
- Dynamic Design: tells how the system behaves and responses to inputs and events, can be expressed with Data flow diagram, Finite State Machine and others
- Example Finite State Machine



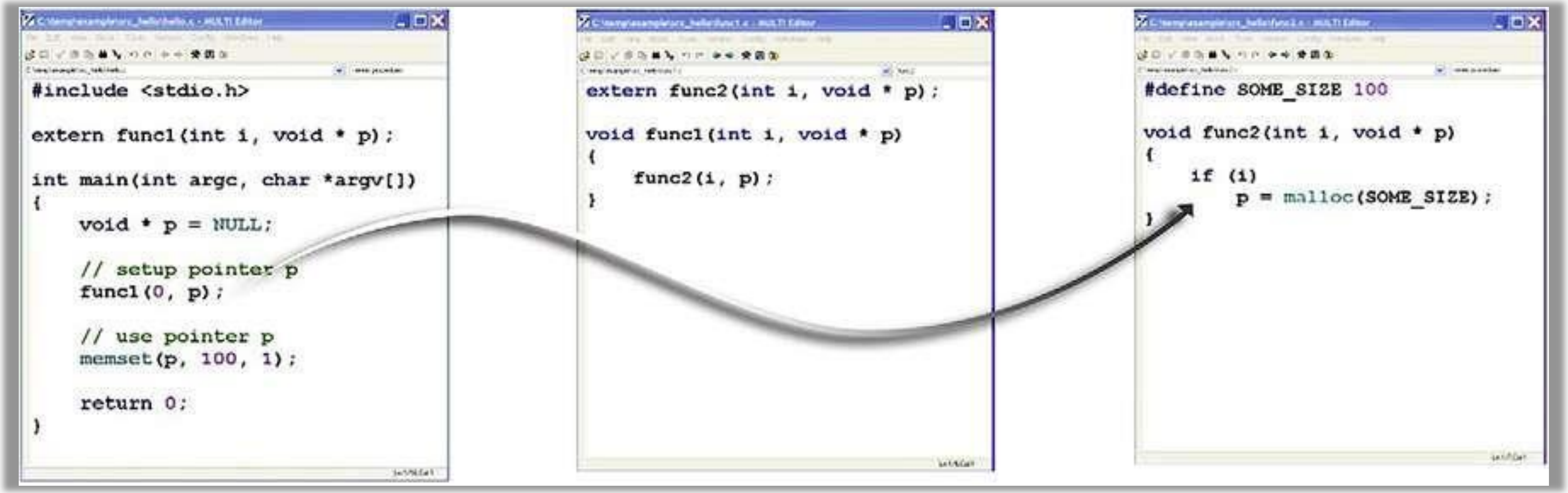
Flow Charts



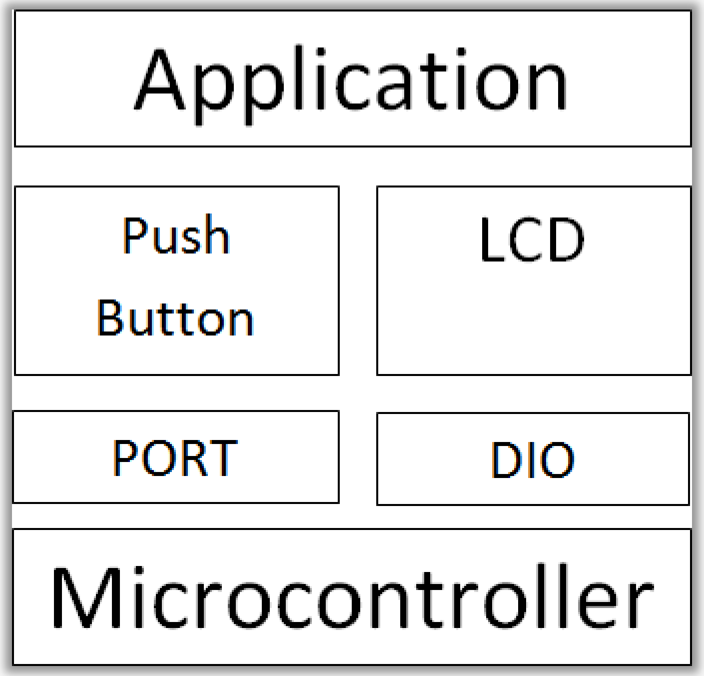
start
No
Btn1
pressed
Yes
Task 1
No
Btn2
pressed
Yes
Task 2
No
Btn3
pressed
Yes
Task 3
No
Btn4
pressed
Yes
Task 4

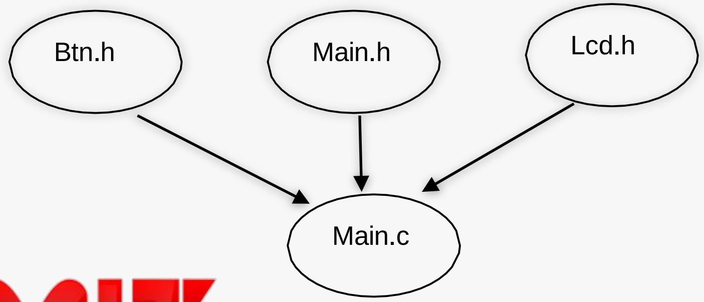
Static Design (Architecture)

Static Design: related to the file structure and functions



Architecture Design





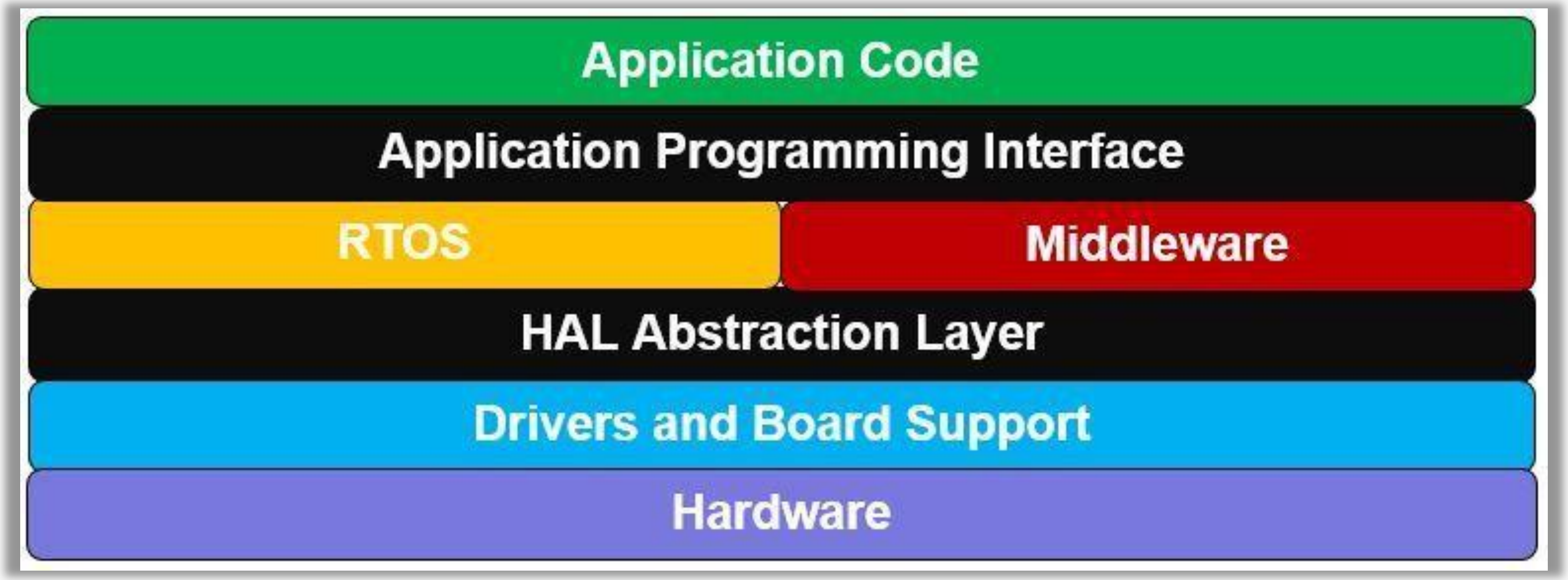
- Benefits
  - Maintainable.
  - Testable.
  - Easy to assign separate "roles"
  - Easy to update and enhance **layers** separately.

File name	***FILE DESCRIPTION***
main.c	The Entry point for the application.
Main.h	Header file contains functions prototypes ,global variables , config.
LCD.h	Header file contains config for LCD
Btn.h	Header file conatins config for Buttons

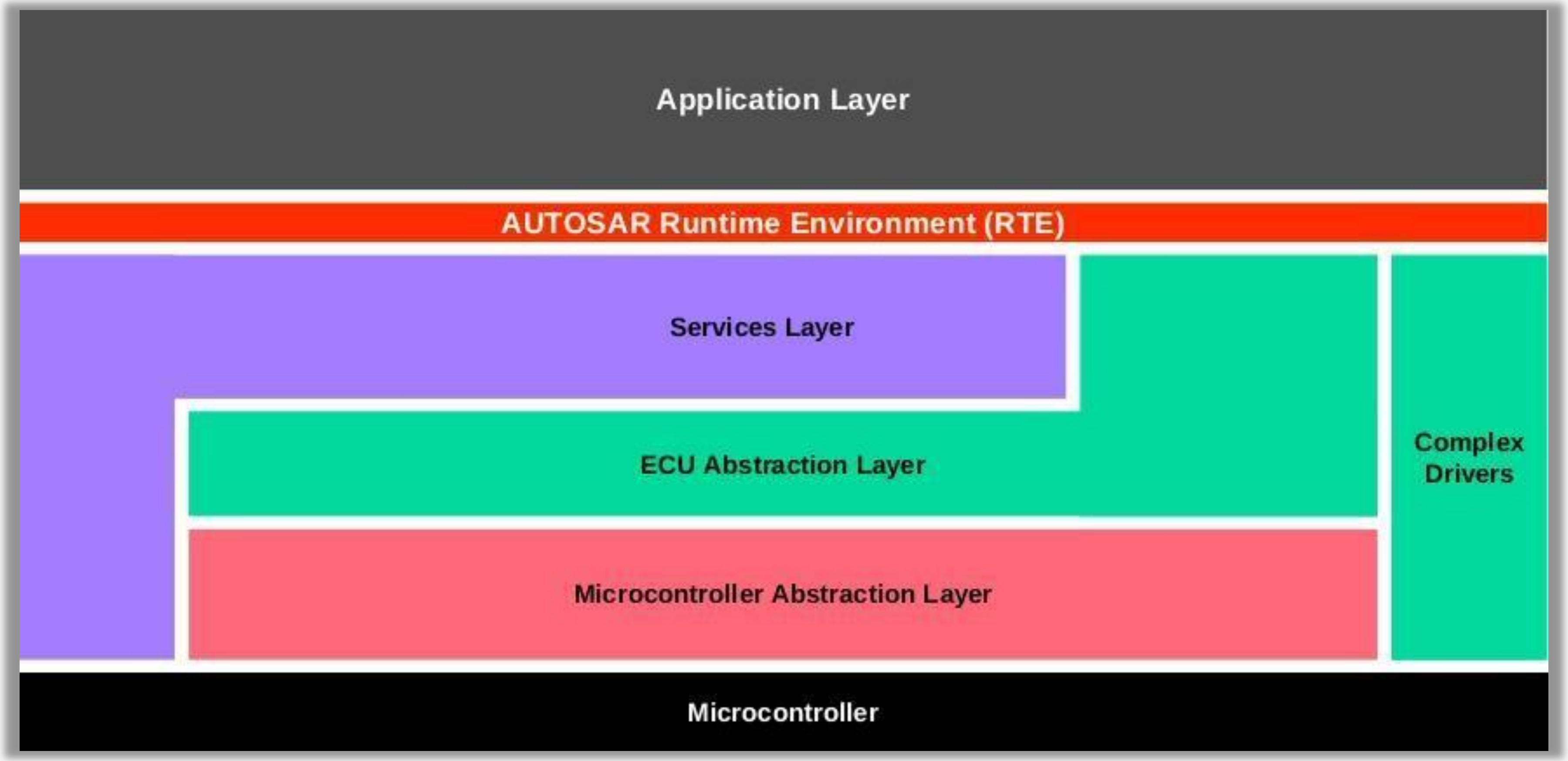
General Layered Arch.

Why Do we need Layered Arch.?

- The Idea is to isolate the Hardware Registers from the Application and other layers of code
- Give the code more flexibility and reusability
- Reduce the complexity during development and integration of various functionalities



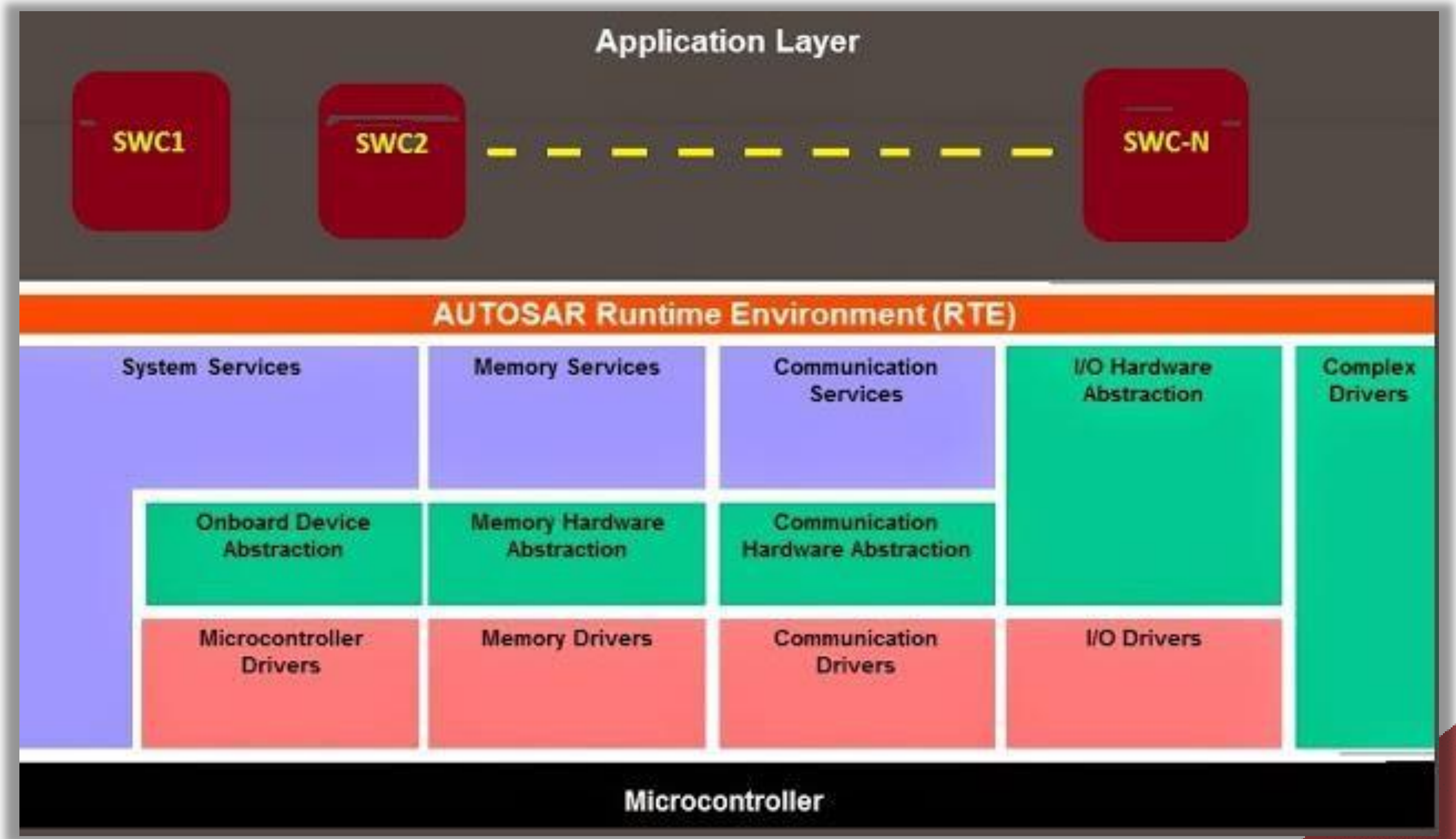
AUTOSAR Basic Architecture



Autosar Basic Architecture

- AUTOSAR Software architecture is broadly classified into 3 categories. They are

1) **Basic Software (BSW)**
2) **Application Software (ASW)**
3) **Run Time Environment (RTE).**



AUTOSAR Stack

- The AUTOSAR has made basic software as layered architecture. The layers are

i)MCAL (Microcontroller Abstraction Layer): Which directly access and control the underlying micro-controller and its resources. It abstracts the micro-controller from its upper layers. [That means the details of registers, memory location, if the controller is a BIG ENDIAN or LITTLE ENDIAN etc are hidden from above layer and provides a uniform interface to them to access the below hardware.]

i)ECUAL (ECU Abstraction Layer): Which interacts with MCAL layer for micro-controller access and directly accesses the hardware resources inside the ECU and not inside micro-controller (like external memory chips, external CAN controllers etc). It abstracts the ECU from its upper layers. If the ECU is changed or upgraded then the layers above ECUAL will not be affected.

AUTOSAR Stack

- iii) Services Layer: Which provides the services to application layers using which the application software components can utilize the hardware resources. Services layer also
includes the operating system. The real time operating system of AUTOSAR is called OSEK.

We visualized BSW as layers (horizontal) before. Basic software can also be viewed as stacks (vertical layers) depending on the functionality. For example The software modules for communication (like CAN Flexray, ethernet etc) in each of the layers like MCAL, ECUAL and services can be visualised as stack of software modules for communication. This is called as comstack. Similarly the AUTOSAR can be visualized as group of following stacks:

i)Service stack: General services (like I2C SPI etc)provided by BSW to upper layer

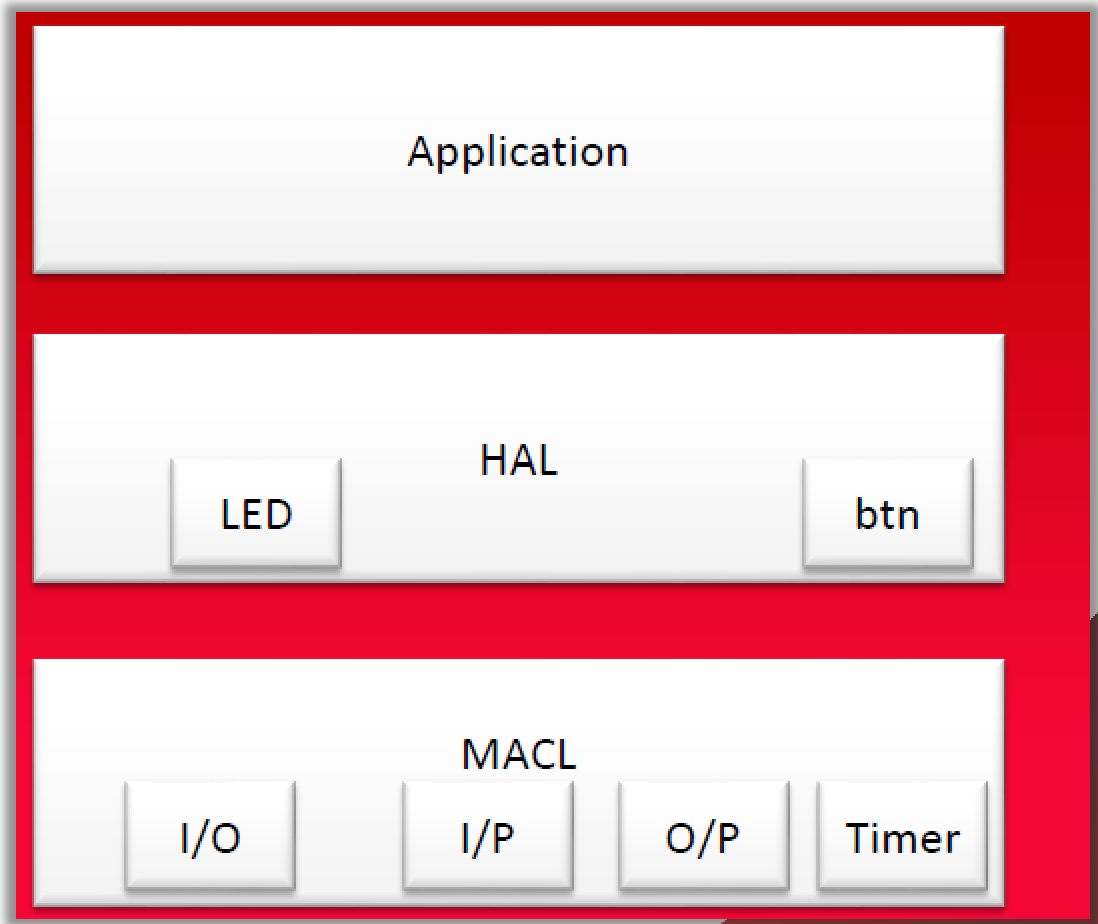
ii) Comstack: Communication stack for intra-ECU or inter-ECU communication
iii) Memstack: Memory stack. Stack of software modules which provide services to access the memory resources of ECU.

iv)Diagstack: Diagnosis stack. Stack of software modules which implement the vehicle diagnosis and tester communication and OBD standards. vehicle diagnostics.]

- v) IO stack: Input Output Stack. The software modules which control the input pins output
pins of ECU, the ADC and other input output related devices present in the ECU.

Example

- In the Previous Slides we showed what we aim to
- Now let’s see how we can start from Ground Up
- Create MCAL Layer with DIO and Timer
- Create HAL Layer with Buttons and LEDS
- Application Code includes the logic which is to ON LED when Button is pressed



References

- https://embedded.fm/blog/2016/6/28/how-big-is-an-enum
- http://www.keil.com/support/man/docs/armclang_intro/armclang_intro_fnb1472741490155.htm
- https://www.rapidtables.com/code/linux/gcc/gcc-o.html
- https://www.engineersgarage.com/ezavr/how-to-write-a-simple-bootloader- for-avr-in-c-language-part-35-46/

References

- https://sites.google.com/site/qeewiki/books/avr-guide/advance-bit-math
- http://users.ece.utexas.edu/~valvano/Volume1/E-Book/C7_DesignDevelopment.htm
- http://www2.phys.canterbury.ac.nz/dept/docs/manuals/unix/DEC_4.0e_Docs/HTML/AQTLTBTE/DOCU_037.HTM
- http://erika.tuxfamily.org/wiki/index.php?title=AUTOSAR-like_DIO_Driver
- http://techietots101.blogspot.com/

AMIT
Beyond Education