System Programming using C

Table of Contents

1. Building an Executable

Unit 1.1 - Linkers and Memory Layouts

Unit 1.2 – Map files

Unit 1.3 – Building Libraries

Key Learning Outcomes

At the end of this module, you will be able to:

1. Explain the importance of linkers and map files.
2. Create their own static library and dynamic library in Linux environment.

UNIT 1.1: Linkers and Memory layouts

Unit Objectives

At the end of this unit, you will be able to:

1. Explain the importance of linkers.

1.1.1 What is QEMU?

The module is designed with the QEMU emulator as the prime development environment. An emulator is a computer program or hardware equipment that can simulate/imitate one computer system by using another system. QEMU can be used to emulate an ARM machine. It is a powerful virtualization and emulation tool that works with a variety of architectures. The code that is written can boot on a real ARM device.

1.1.2 Why QEMU?

We don't have to do the software flashing/download process. No additional hardware is required. We have tools to inspect the state of the emulated hardware. This module uses an Ubuntu-based system for development. But QEMU supports a wide range of systems. QEMU emulator has support for the Texas Instruments platform, which contains Stellaris LM3S6965 Microcontroller. We will write and debug bare metal ARM software for Stellaris LM3S6965 Microcontroller in our example code to understand the module.
"Module3_SystemProgramming_Using_C/BareMetal_LM3S6965" contains the source code files.

1.1.3 QEMU

To install QEMU run the following command in the Ubuntu terminal:

- sudo apt-get install qemu-system-arm

gcc-arm-none-eabi:

- We cannot use the common gcc compiler to build code that executes on an ARM core, instead, we need a cross-compiler. It simply runs on one platform and compiles executables for another platform. We're using Linux on the x86-64 platform, and we want an executable for the ARM platform, so a cross-compiler is used for that. For ARM, the platform gcc-arm-none-eabi toolchain is required.
- To install it run the below command on the terminal:
  - **sudo apt-get install gcc-arm-none-eabi**
- We can verify the installation using commands:

```
akshay@akshayv:BareMetalTest$ qemu-system-arm -version
QEMU emulator version 6.2.0 (Debian 1:6.2+dfsg-2ubuntu6.5)
Copyright (c) 2003-2021 Fabrice Bellard and the QEMU Project developers
akshay@akshayv:BareMetalTest$ arm-none-eabi-gcc --version
arm-none-eabi-gcc (15:10.3-2021.07-4) 10.3.1 20210621 (release)
Copyright (C) 2020 Free Software Foundation, Inc.
This is free software; see the source for copying conditions.  There is NO
warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.

akshay@akshayv:BareMetalTest$
```

Fig 1.1.3 Verifying installation using commands

gdb-multiarch:

- GDB is a source-level debugger, capable of breaking programs at any specific line, displaying variable values, and determining where errors occurred. But GDB is platform specific. To debug our bare-metal program we can use GDB Multiarch.
- To install run the below command:
  - **sudo apt-get install gdb-multiarch**

Stellaris®LM3S6965 Microcontroller Datasheet:

Directory: Module3_SystemProgramming_Using_C/BareMetal_ARM926EJ-S contains all the source code and documentation for LM3D6965 Microcontroller.

1.1.4 Hello World for Bare Metal ARM

LM3D6965 core has three UART serial ports. From the register map of LM3D6965 from the datasheet the address we can find the UART0 is mapped: 0x4000C000

The **UART0** particularly can be used as a terminal when using the "**-nographic**" or **"-serial stdio"** with QEMU

12.5 Register Map

Table 12-3 on page 440 lists the UART registers. The offset listed is a hexadecimal increment to the register's address, relative to that UART's base address:

- UART0: 0x4000.C000
- UART1: 0x4000.D000
- UART2: 0x4000.E000

Note that the UART module clock must be enabled before the registers can be programmed (see page 220). There must be a delay of 3 system clocks after the UART module clock is enabled before any UART module registers are accessed.

Note: The UART must be disabled (see the UARTEN bit in the **UARTCTL** register on page 453) before any of the control registers are reprogrammed. When the UART is disabled during a TX or RX operation, the current transaction is completed prior to the UART stopping.

Table 12-3. UART Register Map

Offset	Name	Type	Reset	Description	See page
0x000	UARTDR	R/W	0x0000.0000	UART Data	442

The code just outputs start_msg and start_msg2 buffers through the device's UART0. The volatile keyword is necessary to instruct the compiler that the memory pointed by lm3s6965_uart0 can change, so don't optimize. The unsigned int type enforces 32-bit read and writes access. Now let's compile the code following the command: arm-none-eabi-gcc -c -mcpu=cortex-m3

-g main.c -o main.o -mcpu flag indicates the processor platform for which the code is compiled. Main.c – Implements the “Hello World!” Program.

```
volatile unsigned int * lm3s6965_uart0 = (unsigned int*)0x4000C000;

void uart0_print(const char* msg);

void main(void)
{
    const char *start_msg = "Hello, World!\r\n";
    const char *start_msg2 = "First Bare-Metal Program ARM :)\r\n";

    uart0_print(start_msg);
    uart0_print(start_msg2);

    while(1);
}

void uart0_print(const char* msg)
{
    while(*msg)
    {
        *lm3s6965_uart0 = *msg;
        msg++;
    }
}
```

1.1.5 C Startup Code

When developing bare metal applications, it is required to supply some functions that we normally take for granted when developing code for mainstream OSs. To write a bare metal program, we must understand the startup sequence of the processor. The ARM Cortex M3 architecture begins to execute code at a determined address, which could be 0x0 the initial value of the stack pointer. The startup code runs before the main() function starts execution.

What is actually needed to start the execution of the main function?

All uninitialized variables are zero. These are stored in the .bss section of the final elf file. All initialized variables are actually initialized. These are stored in the .data section of the final elf file. All static objects are initialized. They may need to get their constructors called if they are not trivial. Function pointers to these static initialization routines are stored in the .init_array section.

The stack pointer is correctly set during startup. Some other machine-dependent features, like enabling access to the floating point coprocessor (VFP coprocessor on most ARM microcontroller architectures etc.). The default startup code is given in the assembly language for the target processor. It makes sense, however, to implement this code in C, for the purposes of creating more generic code that can be used for multiple devices. The startup code does the following tasks.

1.1.6 Vector table for the Nested Vectored Interrupt Controller (NVIC):

Define the vector table for the NVIC. Upon an exception or interruption, it looks up the address of the corresponding ISR.

This table contains:

- The stack pointer and program counter initial values
- The reset vector
- All exception vectors
- All external interrupt vectors
- Fault handlers

When a system reset occurs, execution starts from the reset vector. The processor loads the value of the MSP (main stack pointer) at the highest RAM address (defined by the linker as **__StackTop**). The **__StackTop** variable is actually defined in the linker script. **_sram_stacktop = ORIGIN(SRAM) + LENGTH(SRAM);**

```
/* Exception Table */
__attribute__ ((section(".vectors"), used))
void (* const  _exceptions[CORTEX_M3_EXCEPTIONS])(void) = {
    (void (*)(void))((uint32_t)&_sram_stacktop),          // 00: Reset value of the Main Stack Pointer
    _Reset_Handler,                                       // 01: Reset value of the Program Counter
    _NMI_Handler,                                         // 02: Non-Maskable Interrupt (NMI)
    _Hard_Fault_Handler,                                  // 03: Hard Fault
    _Memory_Mgmt_Handler,                                 // 04: Memory Management Fault
    _Bus_Fault_Handler,                                   // 05: Bus Fault
    _Usage_Fault_Handler,                                 // 06: Usage Fault
    _Unused_Handler,                                      // 07: --
    _Unused_Handler,                                      // 08: --
    _Unused_Handler,                                      // 09: --
    _Unused_Handler,                                      // 10: --
    _SVCall_Handler,                                      // 11: Supervisor Call
    _Debug_Monitor_Handler,                               // 12: Debug Monitor
    _Unused_Handler,                                      // 13: --
    _PendSV_Handler,                                      // 14: Pendable req serv
    _SysTick_Handler,                                     // 15: System timer tick
};
```

Initialize the .data section :

- This section is defined in the linker script with different VMA (Virtual address) and LMA (Load address).
- Since it has to be loaded to Flash, but used from RAM when the execution starts.
- To make sure that all C code can use the initialized data within the .data section, it has to be copied over from Flash to RAM by the startup code.
- To accomplish this it uses the following symbols defined by the linker: **_flash_sdata**, **_sram_sdata** and **_sram_edata.**

Initialize the .bss section to zero:

- It uses the symbols **_sram_sbss** and **_sram_ebss** to obtain the address range that should be set to zero.
- These symbols are defined in the linker script.

_Reset_Handler:

- When a reset occurs it will start executing the code at the address indicated by the _Reset_Handler function pointer.
- This is where we set up and call main().
- We'll create a separate section .startup so this resides immediately after the vector table

```
__attribute__ ((section(".startup")))
void _Reset_Handler(void)
{
  /* Copy the data segment from flash to sram */
  uint32_t *pSrc = &_flash_sdata;
  uint32_t *pDest = &_sram_sdata;

  while(pDest < &_sram_edata)
  {
     *pDest = *pSrc;
     pDest++;
     pSrc++;
  }

  /* Zero initialize the bss segment in sram */
  pDest = &_sram_sbss;

  while(pDest < &_sram_ebss)
  {
     *pDest = 0;
     pDest++;
  }

  /* Call main() */
  main();

  /* main() isn't supposed return
   * - if it does, we need to identify this
   * for now, we'll loop infintely
   */
  while(1);
}
```

"Startup.c" file contains the source code for C Startup.
Compile the Startup Code using the command:

- **arm-none-eabi-gcc -c -mcpu=cortex-m3 -mthumb -g startup.c -o startup.o**

```
akshay@akshayv:BareMetal_LM3S6965$ arm-none-eabi-gcc -c -mcpu=cortex-m3 -mthumb -g main.c -o main.o
akshay@akshayv:BareMetal_LM3S6965$ arm-none-eabi-gcc -c -mcpu=cortex-m3 -mthumb -g startup.c -o startup.o
akshay@akshayv:BareMetal_LM3S6965$ ls
linker.ld  lm3s6965.pdf  main.c  main.o  Makefile  startup.c  startup.o
akshay@akshayv:BareMetal_LM3S6965$
```

1.1.7 Linking Process

What is a linker?

We say “compilation” when referring to the process by which source code is built into an executable file. In previous steps, we compiled the C code: 'main.c' and the assembly code: startup.s. Both those compilations created the object files main.o and startup.o. The linking process comes after compilation. Whether we use an IDE or GCC from the command line, compilation, and linking will happen together as both are invoked with the same command. A linker takes one or more object files, adds external libraries and links it all into an executable file. Each object file is likely to make reference to functions that are held in other object files, and resolving those dependencies is done by the linker. Linker scripts tell the linker how to do the job.

1.1.8 Linking Script

Anatomy of a Linear Script

A linker script holds four things:

Memory layout: The information of what and where the memory is available “**flash**”, and “**ram**” any specific memory available with the CPU (Eg: Core Coupled SRAM, Secondary SRAM, etc.)

Section definitions: Defines which part of a program should go where Data and Code.

Options: Commands to specify architecture, entry point of application if needed

Symbols: Variables to insert into the program at link time

1.1.8.1 Linker Script: Memory Layout

To allocate program space, the linker must know how much memory is available and at what addresses that memory exists. This is defined in the Memory definition in the linker script.

```
Syntax :
MEMORY
 {
   name [(attr)] : ORIGIN = origin, LENGTH = len
   ...
 }
```

The syntax for MEMORY:

name:

Is a name we want to use for the region. Names do not carry meaning, so we are free to use anything we want.

Often “flash” and “ram” as region names.

(attr) :

Are attributes for the region:

writable (w), readable (r), or executable (x).

Flash is (**rx**),

RAM is **rw.**

These attributes describe the properties of the memory, not set it.

origin : Is the start address of the memory region.

len : Is the size of the memory region, in bytes.

TI LM3S6965 Memory Map		
Memory	**Start Address**	**Size**
Flash	0x00000000	256 Kbytes
SRAM	0x20000000	64 Kbytes

This memory map gives us:

```
MEMORY
{
  FLASH     (rx)  : ORIGIN = 0x00000000, LENGTH = 256K
  SRAM      (rw) : ORIGIN = 0x20000000, LENGTH = 64K
}
```

1.1.8.2 Linker Script: Section Definitions

Code and data are bucketed into sections, which are contiguous blocks of memory.

We name those sections as:

- .text - code and constants
- .bss - uninitialized data
- .stack - our stack
- .data - initialized data

.text Section:

The code segment is usually called .text. It contains executable code, which means it’s read-only and has a known size. This section goes in FLASH. The syntax is: This defines a section named .text and

adds it to FLASH. We also need to tell the linker what to put in this section. This is done by listing all of the sections from the input object files we want in .text.

```
SECTIONS
{
   .text :
   {

   } > FLASH
}
```

To find the sections in the object file, we need to look at objdump. To display summary information from the section headers of the object file.

Run the below command on both startup.o and main.o:

arm-none-eabi-objdump -h main.o

arm-none-eabi-objdump -h startup.o

We see that each symbol has a section.

```
akshay@akshayv:BareMetal_LM3S6965$ arm-none-eabi-objdump -h main.o

main.o:     file format elf32-littlearm

Sections:
Idx Name          Size      VMA       LMA       File off  Algn
  0 .text         00000058  00000000  00000000  00000034  2**2
                  CONTENTS, ALLOC, LOAD, RELOC, READONLY, CODE
  1 .data         00000004  00000000  00000000  0000008c  2**2
                  CONTENTS, ALLOC, LOAD, DATA
  2 .bss          00000000  00000000  00000000  00000090  2**0
                  ALLOC
  3 .rodata       00000032  00000000  00000000  00000090  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  4 .debug_info   000000f2  00000000  00000000  000000c2  2**0
                  CONTENTS, RELOC, READONLY, DEBUGGING, OCTETS
  5 .debug_abbrev 000000ac  00000000  00000000  000001b4  2**0
                  CONTENTS, READONLY, DEBUGGING, OCTETS
  6 .debug_aranges 00000020  00000000  00000000  00000260  2**0
                  CONTENTS, RELOC, READONLY, DEBUGGING, OCTETS
  7 .debug_line   00000062  00000000  00000000  00000280  2**0
                  CONTENTS, RELOC, READONLY, DEBUGGING, OCTETS
  8 .debug_str    00000175  00000000  00000000  000002e2  2**0
                  CONTENTS, READONLY, DEBUGGING, OCTETS
  9 .comment      00000034  00000000  00000000  00000457  2**0
                  CONTENTS, READONLY
 10 .debug_frame  00000058  00000000  00000000  0000048c  2**2
                  CONTENTS, RELOC, READONLY, DEBUGGING, OCTETS
 11 .ARM.attributes 0000002d  00000000  00000000  000004e4  2**0
                  CONTENTS, READONLY
akshay@akshayv:BareMetal_LM3S6965$
```

```
akshay@akshayv:BareMetal_LM3S6965$ arm-none-eabi-objdump -h startup.o

startup.o:     file format elf32-littlearm

Sections:
Idx Name          Size      VMA       LMA       File off  Algn
  0 .text         00000006  00000000  00000000  00000034  2**1
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
  1 .data         00000000  00000000  00000000  0000003a  2**0
                  CONTENTS, ALLOC, LOAD, DATA
  2 .bss          00000000  00000000  00000000  0000003a  2**0
                  ALLOC
  3 .vectors      00000040  00000000  00000000  0000003c  2**2
                  CONTENTS, ALLOC, LOAD, RELOC, READONLY, DATA
  4 .startup      00000060  00000000  00000000  0000007c  2**2
                  CONTENTS, ALLOC, LOAD, RELOC, READONLY, CODE
  5 .debug_info   00000140  00000000  00000000  000000dc  2**0
                  CONTENTS, RELOC, READONLY, DEBUGGING, OCTETS
  6 .debug_abbrev 000000cf  00000000  00000000  0000021c  2**0
                  CONTENTS, READONLY, DEBUGGING, OCTETS
  7 .debug_aranges 00000028  00000000  00000000  000002eb  2**0
                  CONTENTS, RELOC, READONLY, DEBUGGING, OCTETS
  8 .debug_ranges 00000018  00000000  00000000  00000313  2**0
                  CONTENTS, RELOC, READONLY, DEBUGGING, OCTETS
  9 .debug_line   000000b9  00000000  00000000  0000032b  2**0
                  CONTENTS, RELOC, READONLY, DEBUGGING, OCTETS
 10 .debug_str    000001c7  00000000  00000000  000003e4  2**0
                  CONTENTS, READONLY, DEBUGGING, OCTETS
 11 .comment      00000034  00000000  00000000  000005ab  2**0
                  CONTENTS, READONLY
 12 .debug_frame  00000048  00000000  00000000  000005e0  2**2
                  CONTENTS, RELOC, READONLY, DEBUGGING, OCTETS
 13 .ARM.attributes 0000002d  00000000  00000000  00000628  2**0
                  CONTENTS, READONLY
akshay@akshayv:BareMetal_LM3S6965$
```

To add all of our functions in the .text section in linker script, we use the syntax: <filename>(<section>).

Filename: The name of the input file whose symbols we want to include in the section.

Section: The name of the input sections

We want the following sections from out files, we can use wildcard *to include from all input files:

- .vector input section, It contains functions we want to keep at the very start of our .text section
- .text
- .startup
- .rodata (segment for constant data)
- .=ALIGN(4);

This means: inserting padding bytes, until the current location becomes aligned on a 4-byte boundary. We will talk about this concept in future chapters.

Finally, the .text section looks like this:

```
.text : {
        KEEP(*(.vectors))
        *(.startup)
        *(.text)
        *(.rodata)
        . = ALIGN(4);
    } > FLASH
```

.stack Section :

Similar to the **.bss** section we do the same with .**stack** memory, since it is in RAM **it** is not loaded.

The stack does not hold symbols, we must explicitly reserve space for it by indicating the size.

```
STACK_SIZE = 0xFA00; /* 64kB */

.stack (NOLOAD) :
{
   . = . + STACK_SIZE;
} > SRAM
```

.data Section:

The .data section holds static variables which have an initial value at the start. Since RAM isn't persisted when power is off, those sections are loaded from flash.

```
.data : {
   . = ALIGN(4);
   *(.data);
   . = ALIGN(4);
} > SRAM AT > FLASH
```

At startup, the Reset_Handler() copies the data from flash to RAM before the main function is called.

To make this viable, every section in our linker script needs two addresses:

Load Address (LMA):

This is where our programmer needs to place the section. LMA is the address "at rest".

Virtual Address (VMA): The VMA is the address during execution i.e., the device is on, and the program is running.

```
Syntax:

.data :
{
   *(.data*);
} > ram AT > rom  /* "> ram" is the VMA, "> rom" is the LMA */
```

Variables:

To make section addresses available to code, the linker is able to create symbols and add them to the program. The syntax is similar to the C assignment: **symbol = expression;**
We need:

- **_etext** - end of code in .text section in FLASH.
- **_sdata** - start of .data section in RAM
- **_edata** - end of .data section in RAM
- **_sbss** - start of .bss section in RAM
- **_ebss** - end of .bss section in RAM
- **_sram_stacktop** - top of the stack.
- **_flash_sdata** - Return the absolute LMA of the **.data** section.

They are straight-forward: we can assign symbols to the value of the location counter (.) at the start and at the end of each section definition.

Final Linker Script:

The Linker script is now ready. You can find the linker script inside: **"linker.ld".**

Now that we have all the files for building and emulating the first Bear Metal App -> **main.c** : main application program, **startup.c** : Startup File & **linker.ld**: Linker File

We already have compiled objects for **main.o** and **startup.o**.

Run the command to create system.elf and system.bin binary image which we can use with QEMU emulator for ARM core.

arm-none-eabi-ld -T linker.ld -o startup.elf startup.o main.o

arm-none-eabi-objcopy -O binary startup.elf system.bin

To run my program in the emulator, the command is:

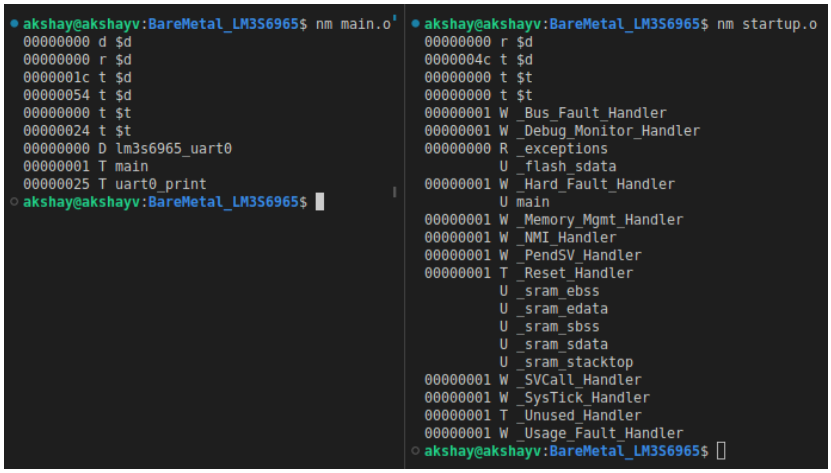
qemu-system-arm -M lm3s6965evb -kernel system.bin -nographic -monitor telnet:127.0.0.1:3456, server,nowait

UART0 particularly can be used as a terminal when using the **-nographic** or "**-serial stdio**" options with QEMU.

The -M option specifies the emulated system.

1.1.8.3 Object File – Symbol Table

In the previous section, we saw that the object files (main.o and startup.o) is the real output from the compilation phase. It's mostly machine code and also contains metadata about the addresses of its functions and variables. These are termed as mbols in a data structure called a “symbol table”. The addresses may not be the final address of the symbol in the final executable. To know the symbols our object files main.o and startup.o contain we can use the linux tool nm (read the link to read about the tool). nm - prints the symbol table for a given object file.



Run the following commands in terminal:

nm main.o

nm startup.o

The output is a list (separated by space) of **address**, **type**, and **symbol name.**

Address: are placeholders in object files, and final in executables.

Name: name of function or variable.

Type:

If **lowercase**, the symbol is usually **local.**

If **uppercase**, the symbol is global (**external**).

"t" The symbol is in the .text section, **"U"** The symbol is undefined, **"D"** The symbol is in the initialized data section, **"W"** The symbol is a weak symbol

UNIT 1.2: Map Files

Unit Objectives

At the end of this unit, students will be able to:

1. Explain the importance of map files.

1.2.1 Map Files

The map file provides important information that helps in understanding and optimizing memory. It is a symbol table for the whole program. A map file provides information about the linker/Locate process, including the following:

- Where object files are mapped into memory.
- How common symbols are allocated?
- All archive members are included in the link, with a mention of the symbol which caused the archive member to be brought in.
- The values are assigned to symbols.

To generate MAP file run the command:

arm-none-eabi-ld -Map output.map -T linker.ld -o output.elf startup.o main.o -Map = *mapfile* Print a link map to the file output.map.

This will generate the file "output.map".

1.2.2 Memory Configuration

This information in the map file are the actual memory regions, along with location, size, and access rights granted to those regions:

```
Memory Configuration

Name             Origin             Length             Attributes
FLASH            0x0000000000000000 0x0000000000040000 xr
SRAM             0x0000000020000000 0x0000000000010000 rw
*default*        0x0000000000000000 0xffffffffffffffff
```

1.2.3 Linker Script and Memory Map

It gives information about symbols in the program. In our file, it indicates the text area size and its content. The interrupt vectors (under the section. vectors) are present at the beginning of the executable, as defined in a startup. These lines give us the address and size of each function. The address of _exceptions is defined under .vector section 0x0000000000000000, coming from startup.o which has a size of 0x40 bytes in the .text area. The size is 0x40 bytes because _exceptions is an array of void * (size = 4 byte) with 16 elements. In this way, we can locate each function used in the program.

```
Linker script and memory map

                0x000000000000fa00                STACK_SIZE = 0xfa00
                0x0000000000000000                . = 0x0

.text           0x0000000000000000      0x134
 *(.vectors)
 .vectors       0x0000000000000000       0x40 startup.o
                0x0000000000000000                _exceptions
 *(.startup)
 .startup       0x0000000000000040       0x60 startup.o
                0x0000000000000040                _Reset_Handler
 *(.text)
 .text          0x00000000000000a0        0x6 startup.o
                0x00000000000000a0                _Debug_Monitor_Handler
                0x00000000000000a0                _SysTick_Handler
                0x00000000000000a0                _Unused_Handler
                0x00000000000000a0                _Hard_Fault_Handler
                0x00000000000000a0                _Bus_Fault_Handler
                0x00000000000000a0                _PendSV_Handler
                0x00000000000000a0                _NMI_Handler
                0x00000000000000a0                _SVCall_Handler
                0x00000000000000a0                _Memory_Mgmt_Handler
                0x00000000000000a0                _Usage_Fault_Handler
 *fill*         0x00000000000000a6        0x2
 .text          0x00000000000000a8       0x58 main.o
                0x00000000000000a8                main
                0x00000000000000cc                uart0_print
```

Initialized variables have to be kept in Flash but they emerge in RAM in the map file, as they are copied into RAM in the .data section before entering
the main(). Symbols _sram_sdata and _sram_edata keep track of the area used in RAM to keep initialized variables. These values are stored in flash, starting at load address 0x0000000000000134. The address of lm3s6965_uart0 defined in main.c is 0x0000000020000000. Similar way .bss section and .stack are defined in the map file.

```
.data           0x0000000020000000        0x4 load address 0x0000000000000134
                0x0000000020000000                . = ALIGN (0x4)
                0x0000000020000000                _sram_sdata = .
 *(.data)
 .data          0x0000000020000000        0x0 startup.o
 .data          0x0000000020000000        0x4 main.o
                0x0000000020000000                lm3s6965_uart0
                0x0000000020000004                . = ALIGN (0x4)
                0x0000000020000004                _sram_edata = .
                0x0000000000000134                _flash_sdata = LOADADDR (.data)

.igot.plt       0x0000000020000004        0x0 load address 0x0000000000000138
 .igot.plt      0x0000000020000004        0x0 startup.o

.bss            0x0000000020000004        0x0 load address 0x0000000000000138
                0x0000000020000004                . = ALIGN (0x4)
                0x0000000020000004                _sram_sbss = .
 *(.bss)
 .bss           0x0000000020000004        0x0 startup.o
 .bss           0x0000000020000004        0x0 main.o
 *(COMMON)
                0x0000000020000004                . = ALIGN (0x4)
                0x0000000020000004                _sram_ebss = .

.stack          0x0000000020000004     0xfa00 load address 0x0000000000000138
                0x000000002000fa04                . = (. + STACK_SIZE)
```

UNIT 1.3: Building Libraries

Unit Objectives

At the end of this unit, students will be able to:

1. Create their own static library and dynamic library in a Linux environment.

1.3.1 Building Libraries

A library is a pre-compiled code that can be reused in an application. Libraries give reusable functions, data structures, and routines that can be reused in code.

1.3.2 Static Libraries

A static library or statically-linked library is a set of external functions, variables and routines, which are resolved in a caller at compile-time. Static libraries are collections of object files that are linked into the stand-alone executable program, with the prefix “lib” and ".a" extension. Static libraries are faster than shared libraries as a set of commonly used object files is put into a single library executable file. Static libraries are bigger in size, as external programs are built into the executable file.

Example code:

To demonstrate static library creation, we have a set of code files inside the directory:
"Module3_SystemProgramming_Using_C/Chapter1_Building_an_Executable/shared-lib"

calculator.c : Library to perform basic math operations.

print_lib.c : Print the terminal characters in color.

header.h : Include file with all function prototypes.

main.c : Application code.

```
akshay@akshayv:shared-lib$ tree
.
├── calculator.c
├── header.h
├── main.c
└── print_lib.c
```

1.3.3 Steps to Generate a Static c Libraries in Linux:

Compile all of our library source code into an object file, GCC does this using command:

- gcc -c calculator.c print_lib.c

We use GNU ar command to create the final library or archive file:

- The archiver, also called ar, is a Unix utility that maintains groups of files as a single archive file.
- ar -cr libDemo.a *.o
  -c : Create the archive.

- -r : updates the library with the most recent version of any existing object file.

Now we can compile the main.c :

- gcc main.c -L. libDemo.a -o main
- -L : flag indicates to the linker that libraries might be found in the current directory (referred to with '.')

Run the main executable.

```
akshay@akshayv:shared-lib$ gcc  -c calculator.c print_lib.c
akshay@akshayv:shared-lib$ ls
 calculator.c  calculator.o  header.h  main.c  print_lib.c  print_lib.o
akshay@akshayv:shared-lib$ ar -cr libDemo.a *.o
akshay@akshayv:shared-lib$ ls
 calculator.c  calculator.o  header.h  libDemo.a  main.c  print_lib.c  print_lib.o
akshay@akshayv:shared-lib$ gcc main.c -L. libDemo.a -o main
akshay@akshayv:shared-lib$ ls
 calculator.c  calculator.o  header.h  libDemo.a  main  main.c  print_lib.c  print_lib.o
akshay@akshayv:shared-lib$ ./main
 4 + 2 = 6
 4 - 2 = 2
 4 * 2 = 8
 4 / 2 = 2
akshay@akshayv:shared-lib$
```

1.3.4 Dynamic Libraries

Also called shared libraries. A shared library or dynamic library is loaded dynamically at runtime for each application that needs it. dynamic linking doesn't require the code to be copied. It is done by placing the name of the library in the binary file. Actual linking is done when the program is executed, both the binary file and the library are in memory.

Example code:

To demonstrate static library creation, we have set of code files inside the directory:

"Module3_SystemProgramming_Using_C/Chapter1_Building_an_Executable/dynamic-lib"

calculator.c : Library to perform basic math operations.

print_lib.c : Print the terminal characters in colour.

header.h : Include file with all function prototypes.

main.c : Application code

```
akshay@akshayv:dynamic-lib$ tree
.
├── calculator.c
├── header.h
├── main.c
└── print_lib.c
```

1.3.5 Steps to Generate a Dynamic c Libraries in Linux

We need to compile all of our library source code into an object file so we inform GCC to do this using the command:

- gcc -c -fPIC calculator.c print_lib.c
- -fPIC flag generates position-independent code, which accesses all constant addresses stored in the Global Offset Table (GOT).

Create a Library file:

It has a format that is specific to the architecture for which the lib is created. We need to use the compiler to create a library. For this we need to inform the compiler that it should create a shared library, not a final program executable file. This is achieved using the '-shared' flag.

- **gcc -shared -o libDemo.so *.o**

Install Library:

The shared library will not be found at runtime if its directory is not in the variable **LD_LIBRARY_PATH.** Type the following command add it:

- **export LD_LIBRARY_PATH=.:$LD_LIBRARY_PATH**

In our example the local directory **('.')** is added to the search. The new path list will be effective only in the shell where you applied the command.

Build the executable:

To create the executable, you first compile the source file:

- **gcc -c main.c**

Then link the created code main.o with the library:

- **gcc -o main main.o -L. -lDemo**

'-L' indicates where the library is to be found

'-l' specifies the library, without the prepending 'lib' and file extension '.so'.

Finally run the executable

```
akshay@akshayv:dynamic-lib$ ls
calculator.c  header.h  main.c  print_lib.c
akshay@akshayv:dynamic-lib$ gcc -c -fPIC calculator.c print_lib.c
akshay@akshayv:dynamic-lib$ ls
calculator.c  calculator.o  header.h  main.c  print_lib.c  print_lib.o
akshay@akshayv:dynamic-lib$ gcc -shared -o libDemo.so *.o
akshay@akshayv:dynamic-lib$ ls
calculator.c  calculator.o  header.h  libDemo.so  main.c  print_lib.c  print_lib.o
akshay@akshayv:dynamic-lib$ export LD_LIBRARY_PATH=.:$LD_LIBRARY_PATH
akshay@akshayv:dynamic-lib$ gcc -c main.c
akshay@akshayv:dynamic-lib$ gcc -o main main.o -L. -lDemo
akshay@akshayv:dynamic-lib$ ls
calculator.c  calculator.o  header.h  libDemo.so  main  main.c  main.o  print_lib.c  print_lib.o
akshay@akshayv:dynamic-lib$ ./main
4 + 2 = 6
4 - 2 = 2
4 * 2 = 8
4 / 2 = 2
akshay@akshayv:dynamic-lib$
```

2. Debugging and Tracing

Unit 2.1 - Debugging with GDB

Unit 2.2 - Trace

Key Learning Outcomes

At the end of this module, you will be able to:

1. Debug the code.
2. Trace the code.

UNIT 2.1: Debugging and Tracing

Unit Objectives

At the end of this unit, you will be able to:

1. Debug the code.

2.1.1 Debugging with GDB

GDB is a debugger for C and C++. GDB offers extensive facilities for tracing and altering the execution of computer programs. The user can monitor and modify the values of programs' internal variables, and even call functions independently of the program's normal behavior.

You can do the following with GDB:

- **Add breakpoints** - Run the program up to a certain point then stop. Print out the values of certain variables at that point. Step through the program one line at a time and print out the values of each variable after executing each line.

GDB is by default typically compiled to target the same architecture as the host system. The host architecture: where the GDB program itself is run. The target architecture: where the program being debugged is run.

In embedded engineering, we often want to target a foreign architecture, e.g., the embedded device, connected to some debug probe. For example, if the target device is **ARM cortex-m3** processor. We need a version of GDB that supports the target architecture of the program being debugged.

gdb-multiarch - contains a version of GDB that supports multiple target architectures.

2.1.1.1 GDB Operation

Compile with the **"-g"** option when using GCC to compile the code.

It generates added information in the object code so the debugger can match a line of source code with the step of execution. Do not use compiler optimization directives such as **"-O"** or **"-O2"** which rearrange computing operations to gain speed as this reordering will not match the order of execution in the source code and it may be impossible to follow. **control+c**: Stop execution. It can stop programs anywhere, in your source or a C library or anywhere. To start the GDB session use the command: **gdb ./<Program Name to debug>** GDB command completion: Use the **TAB** key. Press TAB twice to see all available options if more than one option is available or type.

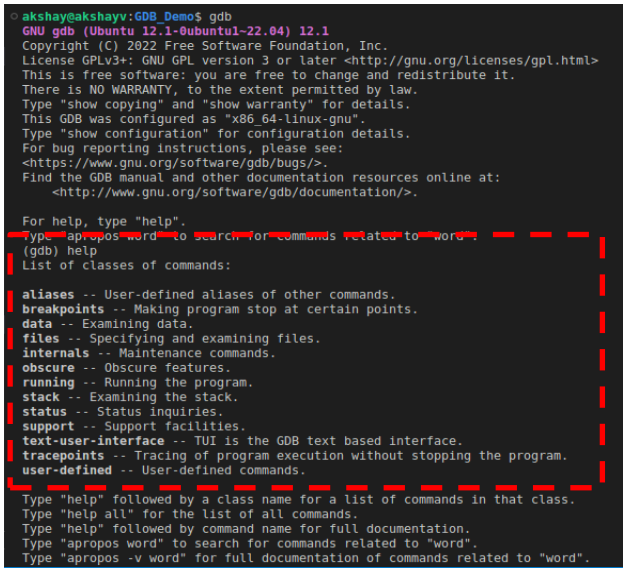
2.1.1.2 GDB Commands

Start and Stop:

Start and Stop	Description
run r run command-line-arguments run < infile > outfile	• Start program execution from the beginning of the program. • The command break main will get you started. • Also allows basic I/O redirection.
continue c c [ignore-count]	• Continue execution to the next break point. • [ignore-count] – Ignore the number of counts breakpoint hits
kill	• Stop program execution.
quit q	• Exit GDB debugger.

Help and Info Commands:

Command	Description
help	List GBD command topics.
help topic-classes	List DDB command within the class.
help command	Command description. Eg help show to list the snow commands.
apropos search-word	Search for commands and command topics containing search-word.
info args i args	List program command line arguments.
info breakpoints	List breakpoints.
info break	List breakpoint numbers.
info break breakpoint-number	List info about specific breakpoints.
info watchpoints	List breakpoints
Info register	List register in use
Info threads	List threads in use
Info set	List set-able option



Breakpoint and Watch:

Break and Watch	Description
break function-name break line-number break ClassName::functionName	Suspend program at the specified function of the line number.
break filename:function	Don't specify a path, just the file name, and function name.
break filename:line-number	Don't specify a path, just the file name, and line number. break Directory/Path/filename.cpp:62
break *address	Suspend processing at an instruction address. Used when you do not have a source.
break line-number if condition	Where the condition is an expression. i.e. x > 10 Suspend when the Boolean expression is true.
break line thread thread-number	Break in a thread at a specified line number. Use info threads to display thread numbers.
tbreak	Temporary break. Break once only. Break is then removed. See "break" above for options.
watch condition	Suspend processing when condition is met. i.e. x > 100
clear clear function clear line-number	Delete breakpoints as identified by command option. Delete all breakpoints in function Delete breakpoints at a given line
delete d	Delete all breakpoints, watchpoints, or catchpoints.

Breakpoint and Watch:

Break and Watch	Description
delete breakpoint-number delete range	Delete the breakpoints, watchpoints, or catch points of the breakpoint ranges specified as arguments.
disable breakpoint-number-or-range enable breakpoint-number-or-range	Does not delete breakpoints. Just enables/disables them. Example: Show breakpoints: info break Disable: disable 2-8
enable breakpoint-number once	Enables once
continue c	Continue executing until the next breakpoint/watchpoint.
continue number	Continue but ignore current breakpoint a number times. Useful for breakpoints within a loop.
finish	Continue bto end of function.

Stack:

Stack	Description
backtrace bt bt inner-function-nesting-depth bt -outer-function-nesting-depth	Show a trace of where you are currently. Which functions you are in. Prints stack backtrace.
backtrace full	Print values of local variables.
frame frame number f-number	Show the current stack frame (a function where you are stopped) Select the frame number. (can also use up/down to navigate frames)
up down up number down number	Move up a single frame (element in the call stack) Move down a single frame Move up/down the specified number of frames in the stack.
info frame	List address, language, address of arguments/local variables, and which registers were saved in the frame.
info args info locals info catch	Info arguments of the selected frame, local variables, and exception handlers.

Line Execution:

- Use to execute code line by line.
- Step into or over a function.
- Run the program till line number.

Line Execution	Description
step s step number-of-steps-to-perform	Step to the next line of code. Will step into a function.
next n next number	Execute the next line of code. Will not enter functions.
until until line-number	Continue processing until you reach a specified line number: Also function name, address, filename:function, or filename:line-number.
info signals info handle handle SIGNAL -NAME option	Perform the following option when signal recieved: nostop, stop, print, noprint, pass/noignore or nopass/ignore
where	Shows current line number and which function you are in.

2.1.1.3 Source Code Commands

To print lines from a source file during the debug session

Source Code	Description
list l list line-number list function list - list start#,end# list filename:function	List source code. To print lines from a source file.
set listsize count show listsize	Number of lines listed when list command given.

Consider sample program: demo_sampleCode.c

The program: Prints n-th Fibonacci Number

Compile the code with the command:

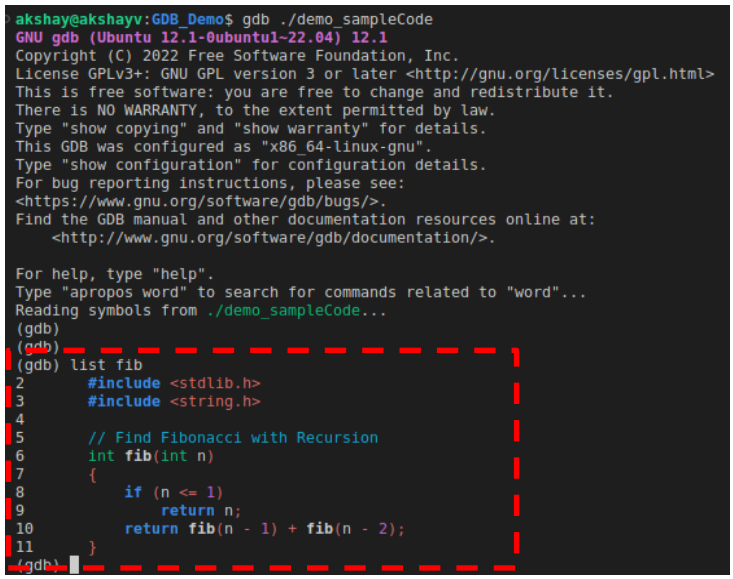
gcc demo_sampleCode.c -g -o demo_sampleCode

Start a GDB Session with the command:

gdb --args ./demo_sampleCode 9

```
akshay@akshayv:GDB_Demo$ gcc demo_sampleCode.c -g -o demo_sampleCode
akshay@akshayv:GDB_Demo$ ./demo_sampleCode  9
Print 9-th Fibonacci Number:
Output : 34
akshay@akshayv:GDB_Demo$
```

This is where we type commands:



Source Code Command:

To view the source code we can uses list command:

To view fib() function use command: list fib

Examples:

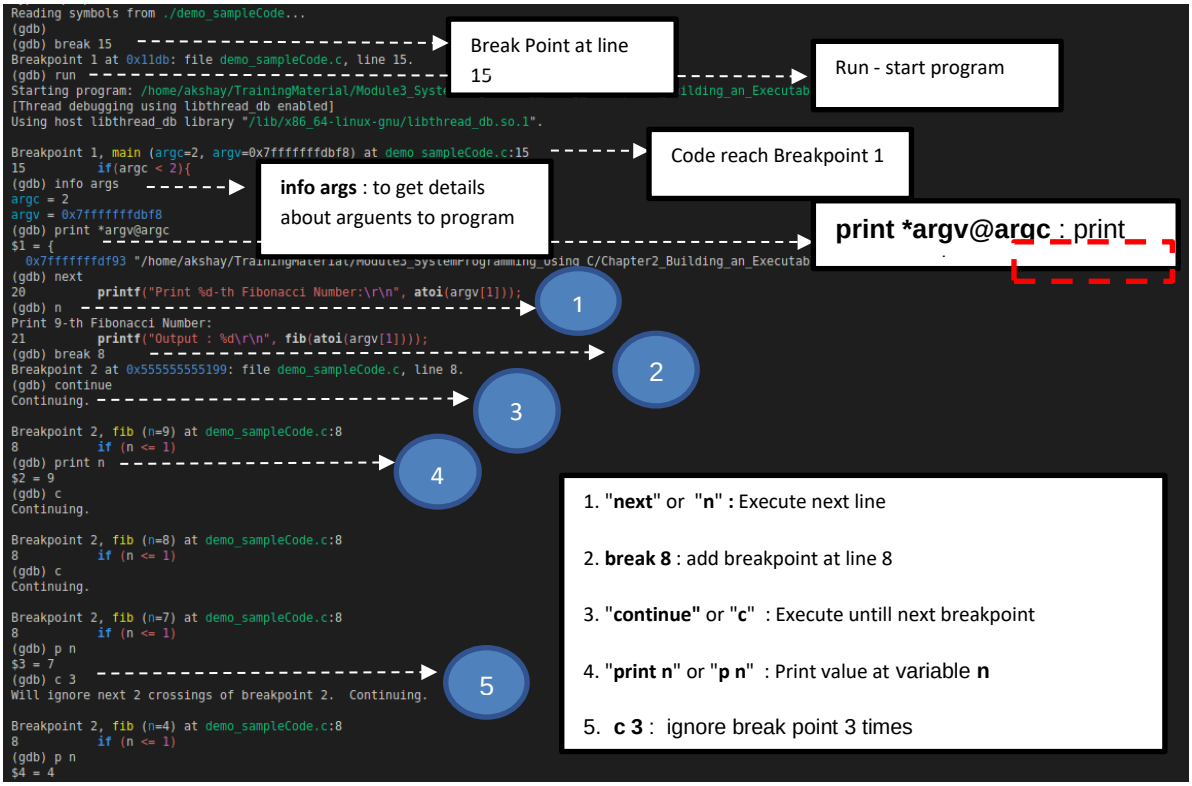


Fig 2.1.1.3 Some GDB commands that can be used to debug the example program

GDB -2 Traverse Call Stack

up: Go up the stack i.e. go to the line that called the function you are currently in.
p n: print value on n (After 2 up commands code is currently in fib(n = 6) function)
down 2: Go down the stack 2 frames
p n: print value on n (After 1 up command and down 2 frames code is currently in fib(n =5) function)

GDB – Watchpoints

Consider the following source code: demo_WatchPoints.c. With GDB we are able to set a watchpoint on a variable in order to break a program when a variable changes. Use display to automatically print how variables change throughout the program's execution.

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main( int argc, char *argv[] )
{
  int count = 0;
  for(int i = 0; i < 4; i++)
  {
    printf("Count Value: %d\r\n", count);
    count += 1;
  }
  return 0;
}
```

```
akshay@akshayv:GDB_Demo$ gcc -g demo_WatchPoints.c -o demo_WatchPoints
akshay@akshayv:GDB_Demo$ ./demo_WatchPoints
Count Value: 0
Count Value: 1
Count Value: 2
Count Value: 3
akshay@akshayv:GDB_Demo$
```

Consider the following source code: demo_sampleSegFault.c
Compile the code with below:
gcc demo_sampleSegFault.c -g -o demo_sampleSegFault
-g flag in order to include appropriate debug information on the binary generated, thus making it possible to inspect it using GDB. When we execute the program, generates a segmentation fault. GDB can be used to inspect the problem.

```
akshay@akshayv:GDB_Demo$ gcc demo_sampleSegFault.c  -g -o demo_sampleSegFault
akshay@akshayv:GDB_Demo$ ./demo_sampleSegFault
Segmentation fault (core dumped)
akshay@akshayv:GDB_Demo$
```

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

size_t get_len( const char *s )
{
  return strlen( s );
}

int main( int argc, char *argv[] )
{
  const char *a = NULL;

  printf( "size of a = %lu\r\n", get_len(a) );

  return 0;
}
```

```
For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from demo_sampleSegFault...
(gdb) run
Starting program: /home/akshay/TrainingMaterial/Module3_SystemProgramming_Using_C/Chapter2_Building_an_Executable/GDB_Demo/demo_sampleSegF
ault
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".

Program received signal SIGSEGV, Segmentation fault.
__strlen_avx2 () at ../sysdeps/x86_64/multiarch/strlen-avx2.S:74
74      ../sysdeps/x86_64/multiarch/strlen-avx2.S: No such file or directory.
(gdb) backtrace
#0  __strlen_avx2 () at ../sysdeps/x86_64/multiarch/strlen-avx2.S:74
#1  0x0000555555555185 in get_len (s=0x0) at demo_sampleSegFault.c:7
#2  0x00005555555551ae in main (argc=1, argv=0x7fffffffdbf8) at demo_sampleSegFault.c:14
(gdb)
```

Get backtrace

The problem is present in **line 7**, and occurs when calling the **strlen(),** because **its argument, s, is NULL**

UNIT 2.2: Trace

Unit Objectives

At the end of this unit, you will be able to:

1. Trace the code.

2.2.1 Trace

Tracing is a technique used to understand what goes on in a running software system. The piece of software used for tracing is called a tracer, which is conceptually similar to a tape recorder. When recording, specific instrumentation points placed in the software source code generate events that are saved on a giant tape: a trace file. You can record user application and operating system events at the same time, opening the possibility of resolving a wide range of problems that would otherwise be extremely challenging. In software engineering, tracing involves the specialized use of logging to record information about a program's execution. This information is typically used by programmers for debugging purposes. This information is used both during development cycles and after the release of the software. Tracing is often compared to logging. However, tracers and loggers are two different tools, serving two different purposes.

Logging vs Tracing:

Logging	Tracing
Consumed primarily by system administrators	Consumed primarily by developers
Logs "high level" information	Logs "low level" information and events that occur much more frequently than log messages
Must not be too "noisy" (containing many duplicate events or information that is not helpful for its intended audience)	Can be noisy
A standards-based output format is often desirable, sometimes even required	Few limitations on the output format
Log messages are often localized	Localization is rarely a concern
Addition of new types of events, as well as new event messages, need not be agile	Addition of new tracing messages must be agile

Software Tracing Techniques:

- Tracing macros.
- Output to debugger.
- Windows software trace preprocessor (aka WPP).
- FreeBSD and SmartOS tracing with DTrace - traces the kernel and the userland.
- Linux kernel tracing with ftrace.
- Linux system-level and user-level tracing with kernel markers and LTTng.
- Linux application tracing with UST is part of the same project as LTTng.
- Linux C/C++ application tracing with cwrap.
- Tracing with GNU Debugger's trace command.

In this module we will go into details of Tracing with LTTng.

2.2.2 Significance of Not Using printf Statements

A common way to do debugging before true debuggers were available was to instrument the code - add strategically placed printf() calls. Thus, progress through the code could be observed. There might also be some kind of pause - a breakpoint really - where the code would be stopped, pending a keystroke or some such user response. A common downside of using printf () calls for debugging is memory usage. The code for this function is quite complex and is really overkill for this usage. Nowadays, that is likely to be less of a problem, but the time taken to execute a call is more likely to be an issue. Not only is the amount of time very likely to be an issue, but the fact is that it is very non-deterministic. Thus, the use of trace tools is very effective for large systems. It can offer highly customizable tracing and profiling capability for single-core and multi-core designs running Linux, Nucleus, or bare metal.

2.2.3 LTTng

The Linux Trace Toolkit: next generation is an open-source software toolkit which you can use to trace the Linux kernel, user applications, and user libraries at the same time.

LTTng consists of:

- Kernel modules to trace the Linux kernel.
- Shared libraries to trace C/C++ user applications.
- Daemons and a command-line tool, lttng, to control the LTTng tracers.

LTTng is currently available on major desktop and server Linux distributions. The main interface for tracing control is a single command-line tool named lttng. The latter can create several recording sessions, enable and disable recording event rules on the fly, filter events efficiently with custom user expressions, start and stop tracing, and much more. In this Module we will focus on User Space applications.

2.2.4 LTTng - Installation

LTTng is a set of software components which interact to instrument the Linux kernel and user applications, and to control tracing.

Those components are bundled into the following packages:

- LTTng-tools

Libraries and command-line interface to control tracing.

- LTTng-modules

Linux kernel modules to instrument and trace the kernel.

- LTTng-UST

Libraries and Java/Python packages to instrument and trace user applications.

We will only need LTTng-tools & LTTng-UST to use the user space LTTng tracer. Install the following dependencies of LTTng-tools and LTTng-UST:

- libuuid -> sudo apt-get install uuid-dev
- Popt -> sudo apt-get install libpopt-dev
- Userspace RCU -> sudo apt-get install liburcu-dev
- Libxml2 -> sudo apt-get install libxml2

  numactl -> sudo apt-get install libnuma-dev

Download, build, and install the latest LTTng-tools 2.13:

wget https://lttng.org/files/lttng-tools/lttng-tools-latest-2.13.tar.bz2 && tar -xf lttng-tools-latest-2.13.tar.bz2 && cd lttng-tools-2.13.* && ./configure && make && sudo make install && sudo ldconfig

Download, build, and install the latest LTTng-UST 2.13:

wget https://lttng.org/files/lttng-ust/lttng-ust-latest-2.13.tar.bz2 && tar -xf lttng-ust-latest-2.13.tar.bz2 && cd lttng-ust-2.13.* && ./configure --disable-numa && make && sudo make install && sudo ldconfig

View and analyze the recorded events:

Babel trace 2 -> sudo apt-get install babeltrace2

By default, LTTng-UST libraries are installed to /usr/local/lib append the path to LD_LIBRARY_PATH: export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/usr/local/lib

2.2.5 LTTng – Core Concepts

2.2.5.1 Core Concepts

From a user's perspective, the LTTng system is built on a few concepts, or objects, on which the lttng command-line tool operates by sending commands to the session daemon (through liblttng-ctl).
The core concepts of LTTng are:

- Instrumentation point, event rule, and event
- Trigger
- Recording session
- Tracing domain
- Channel and ring buffer
- Recording event rule and event record

2.2.5.2 Instrumentation Point

An instrumentation point is a point, within a piece of software, which, when executed, creates an LTTng event. LTTng offers various types of instrumentation depending on the tracing domain:
Linux kernel

- LTTng tracepoint
- Linux kernel system call
- Linux kprobe
- Linux user space probe
- Linux kretprobe

User space
LTTng tracepoint: A statically defined point in the source code of a C/C++ application/library using the LTTng-UST macros.

2.2.5.3 Event Rule and Event

An event rule is a set of conditions to match a set of events. When LTTng creates an event E, an event rule ER is said to match E when E satisfies all the conditions of ER. When an event rule matches an event, LTTng emits the event, therefore attempting to execute one or more actions. As of LTTng 2.13, there are two places where you can find an event rule:

- Recording event rule

A specific type of event rule of which the action is to record the matched event as an event record.

- "Event rule matches" trigger condition (since LTTng 2.13)

When the event rule of the trigger condition matches an event, LTTng can execute user-defined actions such as sending an LTTng notification, starting a recording session, and more.

For LTTng to emit an event E, E must satisfy all the basic conditions of an event rule ER, that is:

The instrumentation point from which LTTng creates E has a specific type. A pattern matches the name of E while another pattern doesn't. The log level of the instrumentation point from which LTTng

creates E is at least as severe as some value, or is exactly some value. The fields of the payload of E and the current context fields satisfy a filter expression.

2.2.5.4 Trigger

A trigger associates a condition to one or more actions. When the condition of a trigger is satisfied, LTTng attempts to execute its actions. As of LTTng 2.13, the available trigger conditions and actions are:

Conditions:

- The consumed buffer size of a given recording session becomes greater than some value.
- The buffer usage of a given channel becomes greater than some value.
- The buffer usage of a given channel becomes less than some value.
- There's an ongoing recording session rotation.
- A recording session rotation becomes completed.
- An event rule matches an event.

Actions:

- Send a notification to a user application.
- Start a given recording session.
- Stop a given recording session.
- Archive the current trace chunk of a given recording session (rotate).
- Take a snapshot of a given recording session.

2.2.5.5 Recording Session

A recording session is a stateful dialogue between you and a session daemon for everything related to event recording. Everything that you do when you control LTTng tracers to record events happens within a recording session. In particular, a recording session:

- Has its own name, unique for a given session daemon.
- Has its own set of trace files, if any.
- Has its own state of activity (started or stopped).
- An active recording session is an implicit recording event rule condition.
- Has its own mode (local, network streaming, snapshot, or live).
- Has its own channels to which are attached their own recording event rules.
- Has its own process attribute inclusion sets.

LTTng offers four recording session modes:

- Local mode: Write the trace data to the local file system.
- Network streaming mode: Send the trace data over the network to a listening relay daemon.
- Snapshot mode: Only write the trace data to the local file system or send it to a listening relay daemon when LTTng takes a snapshot.
- Live mode: Send the trace data over the network to a listening relay daemon for live reading.

Tracing domain

A tracing domain identifies a type of LTTng tracer. A tracing domain has its own properties and features. There are currently five available tracing domains:

- Linux kernel
- User space
- java.util.logging (JUL)
- Log4j
- Python

It is necessary to specify a tracing domain to target a type of LTTng tracer when using some lttng commands to avoid ambiguity. We need to specify a tracing domain when you create an event rule because the Linux kernel and user space tracing domains could have trace points sharing the same name.

2.2.5.6 Channel and Ring Buffer

A channel is an object that is responsible for a set of ring buffers. Each ring buffer is divided into multiple sub-buffers. When a recording event rule matches an event, LTTng can record it to one or more sub-buffers of one or more channels.

When you create a channel, you set its final attributes, that is:

- Its buffering schemes.
- What to do when there's no space left for a new event record because all sub-buffers are full.
- The size of each ring buffer and how many sub-buffers a ring buffer has.
- The size of each trace file LTTng writes for this channel and the maximum count of trace files.
- The periods of its read, switch, and monitor timers.
- For a user space channel: the value of its blocking timeout.

A channel is always associated with a tracing domain. A channel owns recording event rules.

A channel has at least one ring buffer per CPU. LTTng always records an event in the ring buffer dedicated to the CPU that emits it. The buffering scheme of a user space channel determines what has its own set of per-CPU ring buffers.

Per-user buffering: Allocate one set of ring buffers-one per CPU-shared by all the instrumented processes of.

Per-process buffering: Allocate one set of ring buffers-one per CPU-for each instrumented process of.

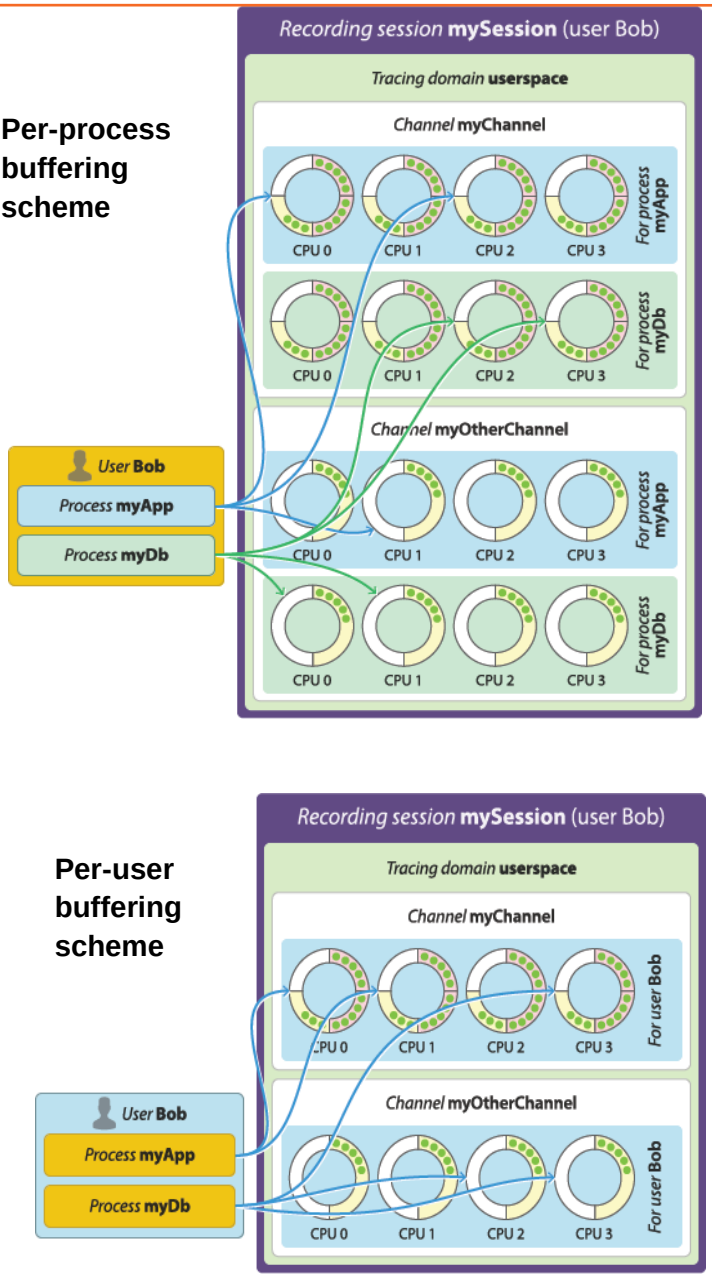


Figure 2.2.5.6 Per-user buffering scheme

The per-process buffering scheme tends to consume more memory than the per-user option because systems generally have more instrumented processes than Unix users running instrumented processes. However, the per-process buffering scheme ensures that one process having a high event throughput won't fill all the shared sub-buffers of the same Unix user, only its own. The buffering scheme of a Linux kernel channel is always to allocate a single set of ring buffers for the whole system.

2.2.5.7 Event Record Loss Mode

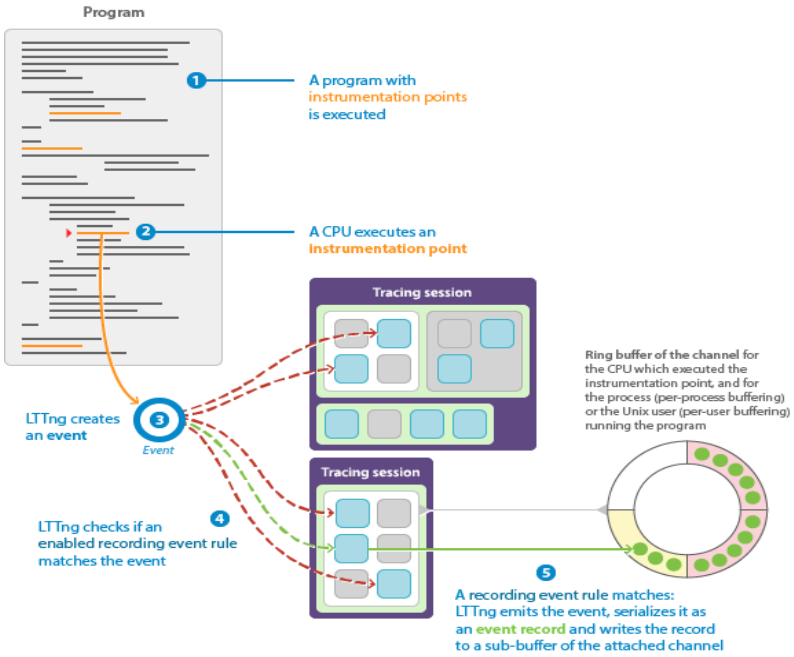
When LTTng emits an event, LTTng can record it to a specific, available sub-buffer within the ring buffers of specific channels. When there's no space left in a sub-buffer, the tracer marks it as consumable and another, available sub-buffer starts receiving the following event records. The available event record loss modes are:

Discard mode: Drop the newest event records until a sub-buffer becomes available.

Overwrite mode: Clear the sub-buffer containing the oldest event records and start writing the newest event records there.

Recording event rule and event record:

A recording event rule is a specific type of event rule in which the action is to serialize and record th e matched event as an event record. You always attach a recording event rule to a channel, which belongs to a recording session, when you create it. When a recording event rule ER matches an event E, LTTng attempts to serialize and record E to one of the available sub-buffers of the channel to which E is attached.



2.2.6 LTTng Tracking – User Application

The high-level procedure to instrument a C or C++ user application with the LTTng user space tracing library, liblttng-ust, is:

Create the source files of a trace point provider package. Add trace points to the source code of the application. Build and link a trace point provider package and the user application. All the Source code for this example is inside: **Chapter2_Building_an_Executable/LTTng_Demo/Hello_World**

Create the source files of a trace point provider package

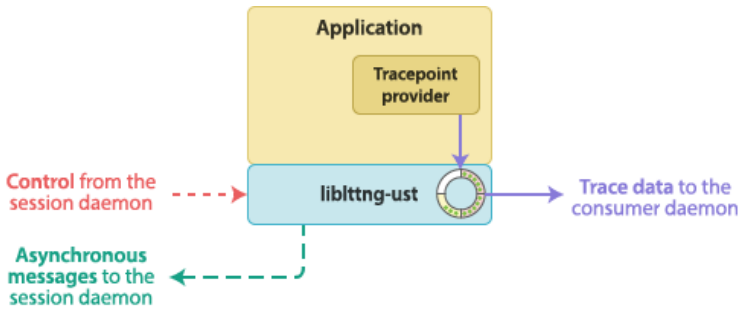
The high-level procedure to instrument a C or C++ user application with the LTTng user space tracing library, liblttng-ust, is:

- Create the source files of a trace point provider package.
- Add trace points to the source code of the application.
- Build and link a trace point provider package and the user application.

All the Source code for this example is inside:
Chapter2_Building_an_Executable/LTTng_Demo/Hello_World

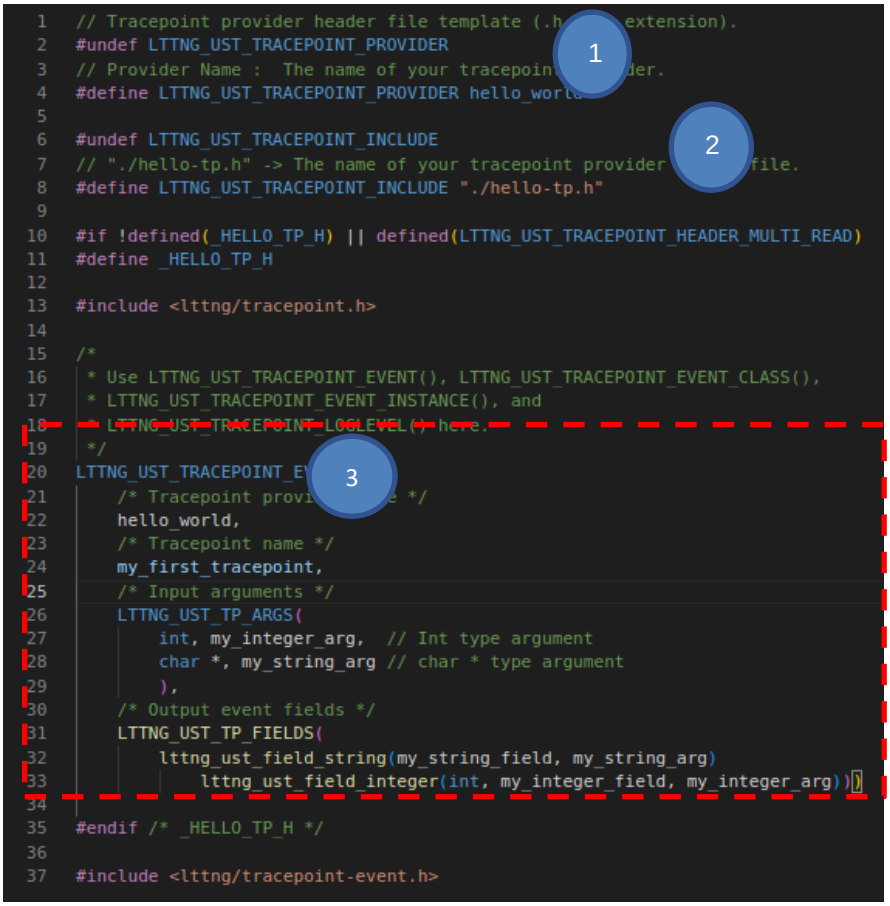
- A trace point provider is a set of compiled functions which provide trace points to an application, the type of instrumentation point which LTTng-UST provides.
- Those functions can make LTTng emit events with user-defined fields (Passing values) and serialize those events as event records to one or more LTTng-UST channel sub-buffers.
- The lttng_ust_tracepoint() macro, which you insert in the source code of a user application, calls those functions.

Its source files are: One or more trace point provider header (.h). A trace point provider package source (.c).



Create a trace point provider header file template

Consider the File: hello-tp.h
provider_name : the name of your trace point provider.
"./hello-tp.h" :The name of your trace point provider header file.
Trace point definition:
Its input arguments LTTNG_UST_TP_ARGS()
They're the macro parameters that the lttng_ust_trace point() macro accepts for this particular trace point in the source code of the user application.
Those arguments can be used in the argument expressions of the output fields defined in LTTNG_UST_TP_FIELDS(). Its output event fields LTTNG_UST_TP_FIELDS()
They're the macro contains the output fields of the trace point, that is, the actual data that can be recorded in the payload of an event emitted by this trace point.
A lttng_ust_field_*() macro specifies the type, size, and byte order of one event field.
Each lttng_ust_field_*() macro takes an argument expression parameter.




```
1  // Tracepoint provider header file template (.h extension).
2  #undef LTTNG_UST_TRACEPOINT_PROVIDER
3  // Provider Name :  The name of your tracepoin der.
4  #define LTTNG_UST_TRACEPOINT_PROVIDER hello_worl
5
6  #undef LTTNG_UST_TRACEPOINT_INCLUDE
7  // "./hello-tp.h" -> The name of your tracepoint provider file.
8  #define LTTNG_UST_TRACEPOINT_INCLUDE "./hello-tp.h"
9
10 #if !defined(_HELLO_TP_H) || defined(LTTNG_UST_TRACEPOINT_HEADER_MULTI_READ)
11 #define _HELLO_TP_H
12
13 #include <lttng/tracepoint.h>
14
15 /*
16  * Use LTTNG_UST_TRACEPOINT_EVENT(), LTTNG_UST_TRACEPOINT_EVENT_CLASS(),
17  * LTTNG_UST_TRACEPOINT_EVENT_INSTANCE(), and
18  * LTTNG_UST_TRACEPOINT_LOGLEVEL() here.
19  */
20 LTTNG_UST_TRACEPOINT_E
21     /* Tracepoint provi e */
22     hello_world,
23     /* Tracepoint name */
24     my_first_tracepoint,
25     /* Input arguments */
26     LTTNG_UST_TP_ARGS(
27         int, my_integer_arg,  // Int type argument
28         char *, my_string_arg // char * type argument
29         ),
30     /* Output event fields */
31     LTTNG_UST_TP_FIELDS(
32         lttng_ust_field_string(my_string_field, my_string_arg)
33             lttng_ust_field_integer(int, my_integer_field, my_integer_arg))
34
35 #endif /* _HELLO_TP_H */
36
37 #include <lttng/tracepoint-event.h>
```

Create a trace point provider package source file

A trace point provider package source file is a C source file which includes a trace point provider header file to expand its macros into event serialization and other functions. Consider the File: hello-tp.c. Add trace points to the source code of an application

Consider the File: hello.c

Once you create a trace point provider header file, use the lttng_ust_trace point() macro in the source code of your application to insert the trace points that this header defines. The lttng_ust_trace point() macro takes at least two parameters:

- The trace point provider name and the trace point name.
- The corresponding trace point definition defines the other parameters.

The demo user app does following things:

- User app first insert the first trace point in app.
- User app inserts the all the arguments passed to user app to trace point along with argument number.
- User inserts a message entry to the trace point.

```
1  #define LTTNG_UST_TRACEPOINT_CREATE_PROBES
2  #define LTTNG_UST_TRACEPOINT_DEFINE
3
4  #include "hello-tp.h"
```

```c
#include <stdio.h>
#include "hello-tp.h"

int main(int argc, char *argv[])
{
    unsigned int i;
    puts("Hello, World!\nPress Enter to continue...");

    /*
     * An lttng_ust_tracepoint() call.
     *
     * Arguments, as defined in `hello-tp.h`:
     *
     * 1. Tracepoint provider name (required)
     * 2. Tracepoint name (required)
     * 3. `my_integer_arg` (first user-defined argument)
     * 4. `my_string_arg` (second user-defined argument)
     */
    lttng_ust_tracepoint(hello_world, my_first_tracepoint, 23,
                         "hi there!");

    for (i = 0; i < argc; i++) {
        lttng_ust_tracepoint(hello_world, my_first_tracepoint,
                             i, argv[i]);
    }

    puts("Quitting now!");
    lttng_ust_tracepoint(hello_world, my_first_tracepoint,
                         i * i, "i^2");
    return 0;
}
```

2.2.7 Record User Application Events

This section walks you through a simple example to record the events of a Hello world program written in C. All the Source code is inside
directory: Chapter2_Building_an_Executable/LTTng_Demo/Hello_World.

Create the traceable user application:

hello-tp.h - The trace point provider header file, which defines the trace points and the events they can generate.

hello-tp.c - The trace point provider package source file.

hello.c - The Hello World application source file:

Build the application:

The trace point provider package: **gcc -c -I. hello-tp.c**

The application: **gcc -c hello.c**

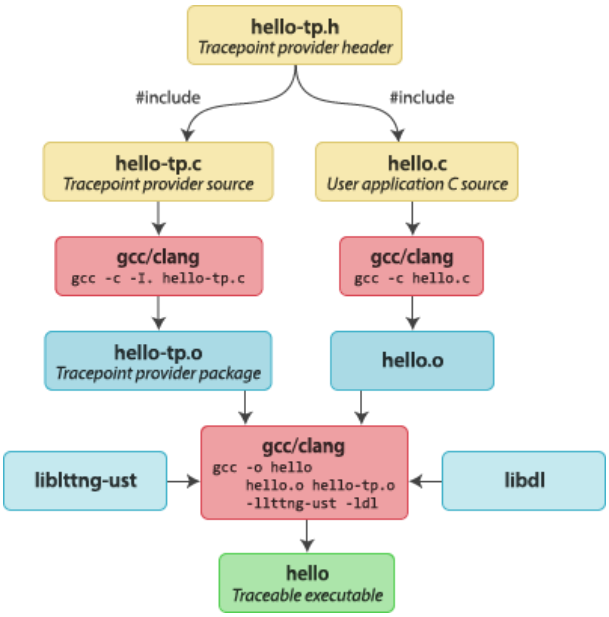
Link the application with the trace point provider package, liblttng-ust and libdl :

gcc -o hello hello.o hello-tp.o -llttng-ust -ldl

Run the application with a few arguments:

./hello world 1 2 3 4 5

```
akshay@akshayv:Hello_World$ ./hello world  1 2 3 4 5
Hello, World!
Press Enter to continue...
```



LTTng start trace session:

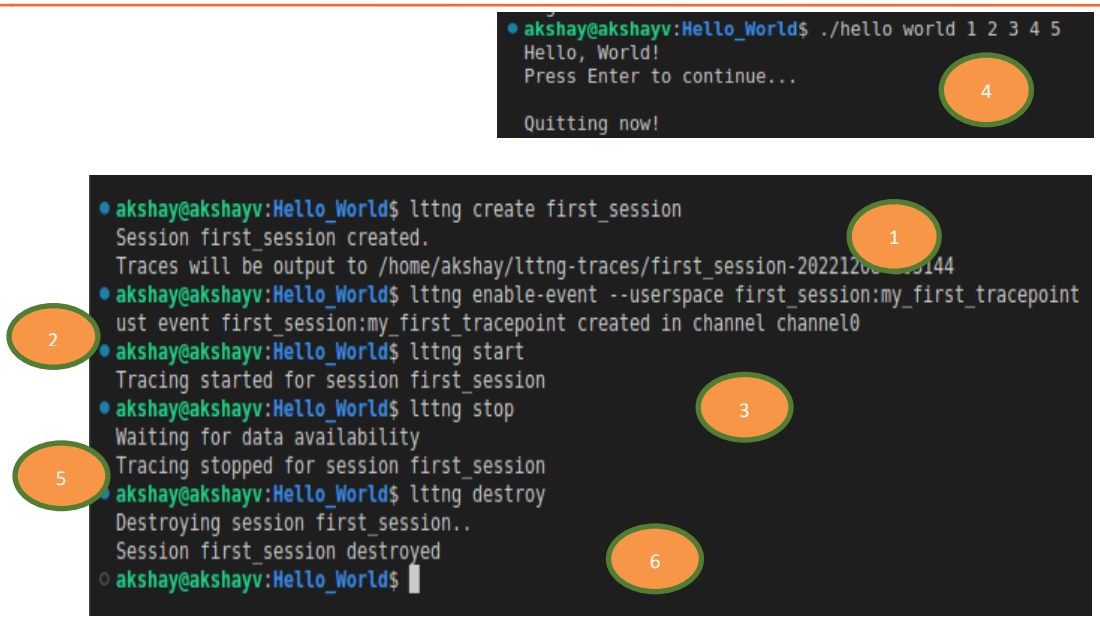
Start an LTTng session daemon: lttng-sessiond –daemonize

Note: A session daemon might already be running, for example as a service that the service manager of your distribution started.

List the available user space trace points: lttng list – user space. In output we see the hello_world:my_first_trace point trace point listed under the ./hello process.

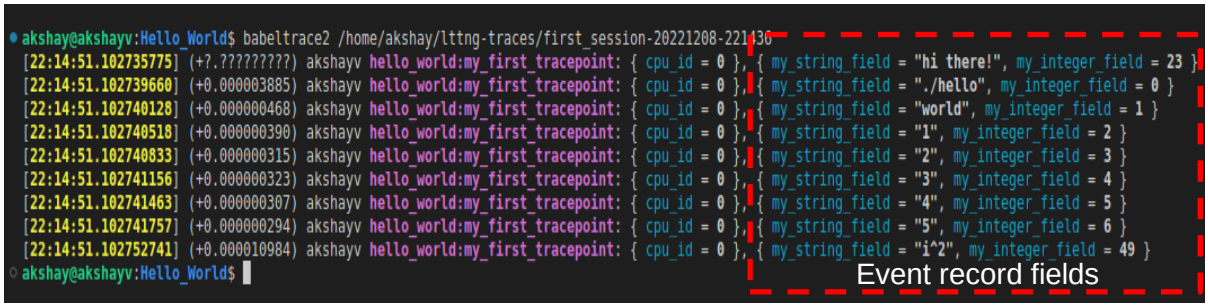
```
akshay@akshayv:Hello_World$ lttng-sessiond –daemonize
Error: A session daemon is already running.
akshay@akshayv:Hello_World$ lttng list --userspace
UST events:
-------------

PID: 284032 - Name: ./hello
      hello_world:my_first_tracepoint (loglevel: TRACE_DEBUG_LINE (13)) (type: tracepoint)
      lttng_ust_lib:unload (loglevel: TRACE_DEBUG_LINE (13)) (type: tracepoint)
      lttng_ust_lib:debug_link (loglevel: TRACE_DEBUG_LINE (13)) (type: tracepoint)
      lttng_ust_lib:build_id (loglevel: TRACE_DEBUG_LINE (13)) (type: tracepoint)
      lttng_ust_lib:load (loglevel: TRACE_DEBUG_LINE (13)) (type: tracepoint)
      lttng_ust_tracelog:LTTNG_UST_TRACEPOINT_LOGLEVEL_DEBUG (loglevel: TRACE_DEBUG (14)) (type: tracepoint)
      lttng_ust_tracelog:LTTNG_UST_TRACEPOINT_LOGLEVEL_DEBUG_LINE (loglevel: TRACE_DEBUG_LINE (13)) (type: tracepoin
t)
```

- Create a recording session: lttng create <session name>
- Create a recording event rule which matches user space tracepoint events named hello_world:my_first_tracepoint: lttng enable-event –userspace hello_world:my_first_tracepoint. Start recording: lttng start .Go back to the running hello application and press Enter.
- The program executes all lttng_ust_tracepoint() instrumentation points, emitting events as the event rule you created matches them, and exits. Stop recording: lttng stop.
- Destroy the current recording session: lttng destroy



View and analyze the recorded events

Once the step **Record user application events** is completed, we can analyze the traces. We will use the **babeltrace2** command-line tool to read the trace. By Default, the traces are saved in directory: **/home/<user>/lttng-traces.** Use command to read the traces: **babeltrace2 ~/lttng-traces/<path to trace directory >.**



2.2.8 LTTng - Log Level

Assign a log level to a trace point definition

Assign a **log level** to a trace point definition with the **LTTNG_UST_TRACEPOINT_LOGLEVEL()** macro. Assigning different levels of severity to trace point definitions can be useful: When you create a recording event rule, you can target trace points having a log level at least as severe as a specific value. The concept of log levels is similar to the levels found in typical logging frameworks, where the given by the functions are used: debug(), info(), warn(), error(), and so on. The syntax of the LTTNG_UST_TRACEPOINT_LOGLEVEL() macro is:

LTTNG_UST_TRACEPOINT_LOGLEVEL(provider_name, trace point_name, log_level)

Where:

provider_name : the trace point provider name.

trace point_name : the trace point name.

log_level : the log level to assign to the trace point definition named trace point_name in the provider_name trace point provider.

While creating a recording event rule matching any user space event from the my_app trace point provider and with a log level range (default channel).

lttng enable-event --userspace my_app:'<trace point name>' --loglevel=INFO

2.2.9 LTTng – API's

lttng_ust_tracef()

It is a small LTTng-UST API designed for quick, **printf**(3)-like instrumentation without the burden of creating and building a trace point provider package. To use lttng_ust_tracef() in your application:
In the C or C++ source files where you need to use lttng_ust_tracef(), include <lttng/tracef.h>:
#include <lttng/tracef.h>
In the source code of the application, use **lttng_ust_tracef()** like you would use **printf**:
lttng_ust_tracef("my message: %d (%s)", my_integer, my_string);
Link your application with liblttng-ust:
gcc -o app app.c -llttng-ust
Create a recording event rule which matches user space events named **lttng_ust_tracef:***:
lttng enable-event --userspace 'lttng_ust_tracef:*'
lttng_ust_tracef() is useful for some quick prototyping and debugging, but you shouldn't consider it for any permanent and serious applicative instrumentation.

lttng_ust_tracelog()

API is very similar to lttng_ust_tracef(), with the difference that it accepts an **additional log level** parameter.
The goal of lttng_ust_tracelog() is to ease the migration from logging to tracing.
To use lttng_ust_tracelog() in your application:
In the C or C++ source files where you need to use tracelog(), include <lttng/tracelog.h>:
<lttng/tracelog.h>
In the source code of the application, use lttng_ust_tracelog() like you would use **printf**(3), except for the first parameter which is the log level:
tracelog(LTTNG_UST_TRACEPOINT_LOGLEVEL_WARNING, "my message: %d (%s)", my_integer, my_string);
Link your application with liblttng-ust:
gcc -o app app.c -llttng-ust
Create a recording event rule which matches user space trace point events named lttng_ust_tracelog:* and with some minimum level of severity:
lttng enable-event --userspace 'lttng_ust_tracelog:*' --loglevel=WARNING.

2.2.10 LTTng – Example

This example shows the features we saw in the previous sections. The static linking method is chosen here to link the application with the trace point provider. This App shows demo of how to use trace point events in the user app.

Follow the command sequence to build and run example

```
akshay@akshayv:Example_App$ gcc -c -I. myTp.c
akshay@akshayv:Example_App$ gcc -c app.c
akshay@akshayv:Example_App$ gcc -o app myTp.o app.o -llttng-ust -ldl
akshay@akshayv:Example_App$ lttng create my-session
Session my-session created.
Traces will be output to /home/akshay/lttng-traces/my-session-20221209-220419
akshay@akshayv:Example_App$ lttng enable-event --userspace 'my_provider:*'
ust event my_provider:* created in channel channel0
akshay@akshayv:Example_App$ lttng start
Tracing started for session my-session
akshay@akshayv:Example_App$ ./app My Arguments 1 2 3 4 5
akshay@akshayv:Example_App$ lttng stop
Waiting for data availability
Tracing stopped for session my-session
akshay@akshayv:Example_App$ lttng view
Trace directory: /home/akshay/lttng-traces/my-session-20221209-220419

[22:04:51.324163298] (+?.??????????) akshayv my_provider:simple_event: { cpu_id = 1 }, { argv = "./app", argc = 8 }
[22:04:51.324165231] (+0.000001933) akshayv my_provider:simple_event: { cpu_id = 1 }, { argv = "./app", argc = 0 }
[22:04:51.324165671] (+0.000000440) akshayv my_provider:simple_event: { cpu_id = 1 }, { argv = "My", argc = 1 }
[22:04:51.324166085] (+0.000000414) akshayv my_provider:simple_event: { cpu_id = 1 }, { argv = "Arguments", argc = 2 }
[22:04:51.324166509] (+0.000000424) akshayv my_provider:simple_event: { cpu_id = 1 }, { argv = "1", argc = 3 }
[22:04:51.324166805] (+0.000000296) akshayv my_provider:simple_event: { cpu_id = 1 }, { argv = "2", argc = 4 }
[22:04:51.324167096] (+0.000000291) akshayv my_provider:simple_event: { cpu_id = 1 }, { argv = "3", argc = 5 }
[22:04:51.324167370] (+0.000000274) akshayv my_provider:simple_event: { cpu_id = 1 }, { argv = "4", argc = 6 }
[22:04:51.324167661] (+0.000000291) akshayv my_provider:simple_event: { cpu_id = 1 }, { argv = "5", argc = 7 }
[22:04:51.324235477] (+0.000067816) akshayv my_provider:big_event: { cpu_id = 1 }, { int_field1 = 20, stream_pos = 0xA, float_field
 = -3.14, string_field = "hello tracepoint", array_field = [ [0] = 100, [1] = -35, [2] = 1, [3] = 23, [4] = 14, [5] = -6, [6] = 28,
 [7] = 1001, [8] = -3000 ] }
[22:04:51.324302093] (+0.000066616) akshayv my_provider:event_instance1: { cpu_id = 1 }, { a = 23, b = 8, c = "App at Stage : 1" }
[22:04:51.324302777] (+0.000000684) akshayv my_provider:event_instance2: { cpu_id = 1 }, { a = 17, b = 40, c = "App at Stage : 2" }
[22:04:51.324303310] (+0.000000533) akshayv my_provider:event_instance3: { cpu_id = 1 }, { a = -52, b = 23, c = "Closing App" }
akshay@akshayv:Example_App$
```

```
akshay@akshayv:Example_App$ tree
.
├── app.c
├── app.h
├── myTp.c
└── myTp.h

0 directories, 4 files
akshay@akshayv:Example_App$
```

Figure 2.2.9 View and analyze the recorded events

3. Memory Management

Key Learning Outcomes

At the end of this module, you will be able to:

1. Explain the memory architecture of code with example.
2. Explain the memory allocation of variables with example.
3. Analyze the static code and memory profiling using suitable tool.

UNIT 3.1: Memory Architecture

Unit Objectives

At the end of this unit, you will be able to:

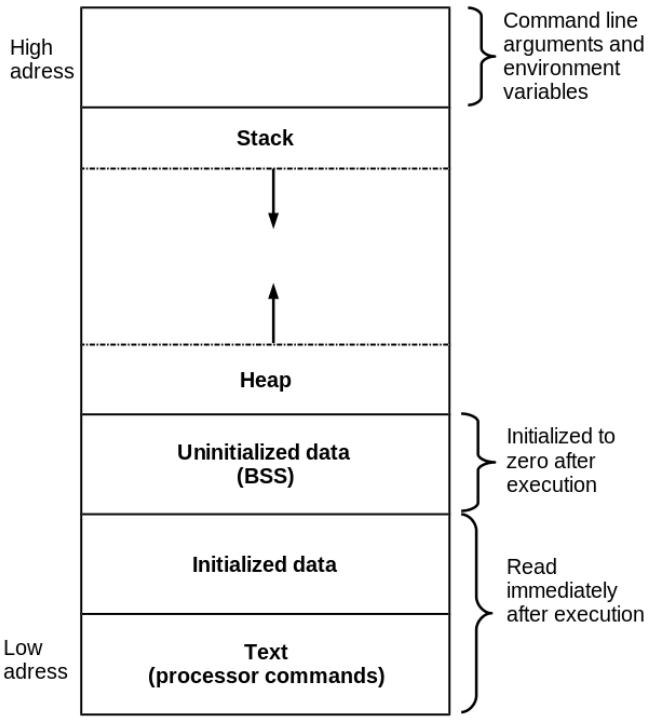
1. Explain the memory architecture of code with example

3.1.1 Memory Architecture

Once a C program is compiled, a binary executable file is created. On execution of the program, this binary executable file loads into RAM. As the access time for secondary storage is longer than RAM, computers access instructions from RAM. Though RAM is faster, programmers should utilize it efficiently as it has a limited storage capacity and is expensive. The memory layout of C program helps programmers to decide the amount of memory used by the program for execution.

A typical memory representation of a C program comprises the following sections:

- Text segment (Instructions or Processor commands)
- Initialized data segment
- Uninitialized data segment (bss)
- Stack
- Heap



3.1.2 Text Segment

Text segment, also called a code segment or simply text. The compiled program is converted to executable binary file, it contains instructions, these instructions/processor commands get stored in the text segment. Text segment is placed below the heap or stack area, this prevents heaps and stack overflows from overwriting it. text segment is a read-only area, this prevents a program from accidentally modifying its instructions.

3.1.2 Initialized Data Segment

Initialized data segment, simply called the Data Segment. Data segment is an area of the virtual address space of a program. It comprises all the constants, both global and static variables. These are initialized at the time of variable declaration in the program. These variable values can be change during program execution. This memory area has read-write permission. Further it is classified into the read-write and read-only areas. const variable goes under the read-only area. All other variables in this segment come in the read-write area.

```c
#include <stdio.h>

/* global variables stored in the read-write part of
   initialized data segment
 */
int global_var = 50;
char *hello = "Hello World";

/* global variables stored in the read-only part of
   initialized data segment
 */
const int global_var2 = 30;

int main()
{
    // static variable stored in initialized data segment
    static int a = 10;
    // ...
    return 0;
}
```

3.1.2.2 Uninitialized Data Segment (bss)

Uninitialized data segment, also called the "**bss**" segment. The kernel initializes the Data from this segment to arithmetic 0 before the program starts executing. Uninitialized data starts at the end of the initialized data segment. It comprises all uninitialized global variables and static variables. This data segment has **read-write permissions**.

```c
#include <stdio.h>

// Uninitialized global variable stored in the bss segment
int global_variable;

int main()
{
    // Uninitialized static variable stored in bss
    static int static_variable;

    // ..
    printf("global_variable = %d\n", global_variable);
    printf("static_variable = %d\n", static_variable);
    return 0;
}
```

```
Output:

global_variable = 0
static_variable = 0
```

3.1.4 Stack

The stack area is traditionally above the heap area and grows in the downward direction (opposite to heap). On x86 based computer, it grows toward address zero. On some other architectures, it grows in the opposite direction. The stack is LIFO (last-in-first-out) data structure. A "stack pointer (SP)" register points the top of the stack. Each time a new value is "pushed" onto stack, SP is adjusted. A stack frame is a set of values pushed for a function call. A stack frame comprises values of local variables, parameters passed to a function, some of the machine registers & return address. For each function call new stack frame is created.

```c
#include<stdio.h>

void foo() {
    // local variables stored in the stack
    // when the function call is made
    int var1, var2;
}

int main() {
    // local variables stored in the stack
    int count = 5;
    char name[26];
    foo();
    // ..
    return 0;
}
```

How stack looks?

Consider Program: stack_growth_example.c

- main_local - local variable inside main.
- func() - has its own local variable func_local.
- Call the func() from main().
- Compare addresses of two local variable.
- If address of func()'s local variable is greater than main's local variable, the stack grows upward (addresses increase).
- Inside func() increment the function local variable and call the func() recursively.
- This will eventually grow the stack and cause the stack overflow and finally core dump.
- Let's use GDB debugger to see how the stack looks.

```c
void func(int *arg_addr)
{
    // Function Local Variable :
    static int func_local = 2;
    // Compare address of argument with
    // address function local variable
    if (arg_addr < &func_local)
        printf("Stack grows upward\n");
    else
        printf("Stack grows downward\n");

    // Incr the variable:
    func_local++;
    // Call the function again to grow stack:
    func(&func_local);
}

int main()
{
    // Main local variable
    int main_local = 1;

    func(&main_local);
    return 0;
}
```

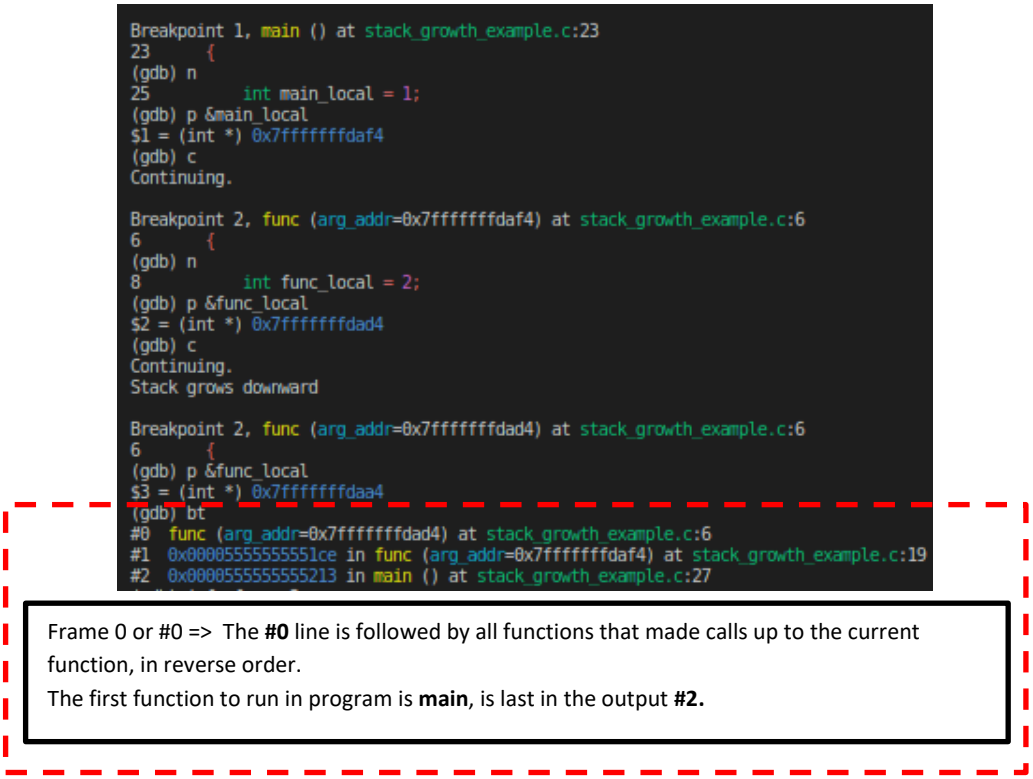
```
Stack grows downward
Stack grows downward
Stack grows downward
Stack grows downward
Stack grows downward
Stack grows downward
Stack grows downward
Stack grows downward
Stack grows downward
Segmentation fault (core dumped)
akshay@akshayv:Chapter3_Memory_Management$
```

Run GDB debugger with command:

gdb ./stack_growth_example

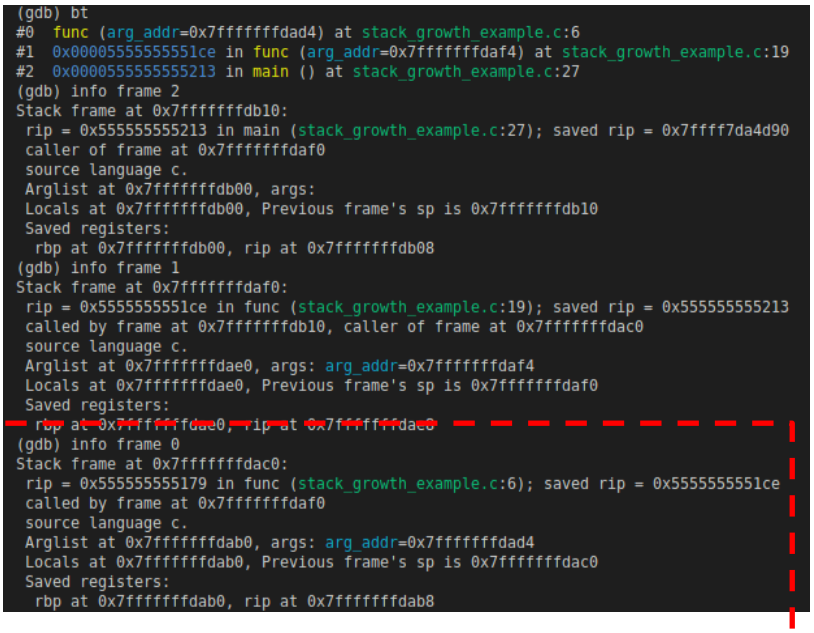
We can also use below commands to analyze the stack:

It can be useful, for e.g., in analysis of stack overflow. Backtrace lists the functions on the stack. Select-frame n it is variant of frame command, it does not display the new frame after selecting it. Info frame allows to print different stack frames to be selected.



Frame 0 or #0 => The **#0** line is followed by all functions that made calls up to the current function, in reverse order.

The first function to run in program is **main**, is last in the output **#2.**



(gdb) info frame 0

Stack frame at 0x7fffffffdac0: This 0x7fffffffdac0 is the stack address allocated by your current program

rip = 0x555555555179 in func (stack_growth_example.c:6); saved rip = 0x5555555551ce

0x555555555179 it is the address of called function, 0x5555555551ce is the address of the code calling the function, namely: 0x5555555551ce: call 0x555555555179

called by frame at 0x7fffffffdaf0

source language c.

Arglist at 0x7fffffffdab0, args: arg_addr=0x7fffffffdad4

arglist is the address 0x7fffffffdab0 of the function parameter in the stack

Locals at 0x7fffffffdab0,

Address of local variables.

Previous frame's sp is 0x7fffffffdac0

This is where the caller frame´s stack pointer points to, at the moment of calling, it is also the start memory address of called stack frame.

Saved registers:

rbp at 0x7fffffffdab0, rip at 0x7fffffffdab8

These two addresses are on the callee stack, for two saved registers.

rbp: is to stash the value of the stack pointer at the start of the function.

rip: it contains the address of the next instruction to execute. When the instruction is executed, the eip will be pointing to the next instruction in the memory.

In our case rbp is 8 smaller than eip, this is because we have an argument arg_addr which takes 4 bytes.

3.1.4 Heap

The heap is the area where dynamic memory allocation takes place. The heap generally begins at the end of bss segment. It grows and shrinks in the opposite direction of the Stack. It is managed by functions like **malloc/new**, **free/delete**, which use the **brk** and **sbrk** system calls to adjust its size. Heap data segment is used among modules loading dynamically and all the shared libraries in a program.

The allocation to the heap area occurs when:

- Memory size is dynamically allocated at run-time.
- Scope is not limited. (Variables referenced from several places).
- Memory size is large.

```c
#include <stdio.h>
#include <stdlib.h>
int main() {
    // memory allocated in heap segment
    char *var = (char*) malloc ( sizeof(char) );
    // ..
    return 0;
}
```

The Linux command **size** reports the size (in bytes) for the text, data, and bss segments.

```c
#include <stdio.h>

int main(void)
{
    return 0;
}
```

Simple program

```
TERMINAL
akshay@akshayv:Chapter3_Memory_Management$ gcc test_memory_layout.c -o test_memory_layout
akshay@akshayv:Chapter3_Memory_Management$ size test_memory_layout
   text    data     bss     dec     hex filename
   1228     544       8    1780     6f4 test_memory_layout
akshay@akshayv:Chapter3_Memory_Management$
```

```c
#include <stdio.h>

int Global; /* Uninitialized variable stored in bss*/

int main(void)
{
    static int i; /* Uninitialized static variable stored in bss */
    return 0;
}
```

```
TERMINAL
akshay@akshayv:Chapter3_Memory_Management$ gcc test_memory_layout.c -o test_memory_layout
akshay@akshayv:Chapter3_Memory_Management$ size test_memory_layout
   text    data     bss     dec     hex filename
   1228     544      16    1788     6fc test_memory_layout
akshay@akshayv:Chapter3_Memory_Management$
```

Simple C program

```c
#include <stdio.h>

int Global; /* Uninitialized variable stored in bss*/

int main(void)
{
    static int i; /* Uninitialized static variable stored in bss */
    static int Count = 100; /* Initialized static variable stored in data */
    return 0;
}
```

```
TERMINAL
akshay@akshayv:Chapter3_Memory_Management$ gcc test_memory_layout.c -o test_memory_layout
akshay@akshayv:Chapter3_Memory_Management$ size test_memory_layout
   text    data     bss     dec     hex filename
   1228     548      12    1788     6fc test_memory_layout
akshay@akshayv:Chapter3_Memory_Management$
```

3.1.5 Memory Allocation

Memory allocation is a method by which computer programs and services are assigned with physical or virtual memory space. The memory allocation is either before or at the time of program execution.

In C memory is allocated in:

- Stack memory
- Heap memory

When memory is allocated at compile time it is stored in stack area. When memory is allocated at runtime, it is stored in heap area.

Types of memory allocations:

- Compile-time or Static Memory Allocation.
- Run-time or Dynamic Memory Allocation.

3.1.5.1 Stack Memory Area

A stack is a linear data structure. Access time is fast. Space is managed efficiently by OS, so memory will never be fragmented. Variables cannot be resized. Memory allocated in a contiguous block. Allocation and deallocation done by compiler. You should use the stack when you work with relatively small variables that required until the function using them is alive.

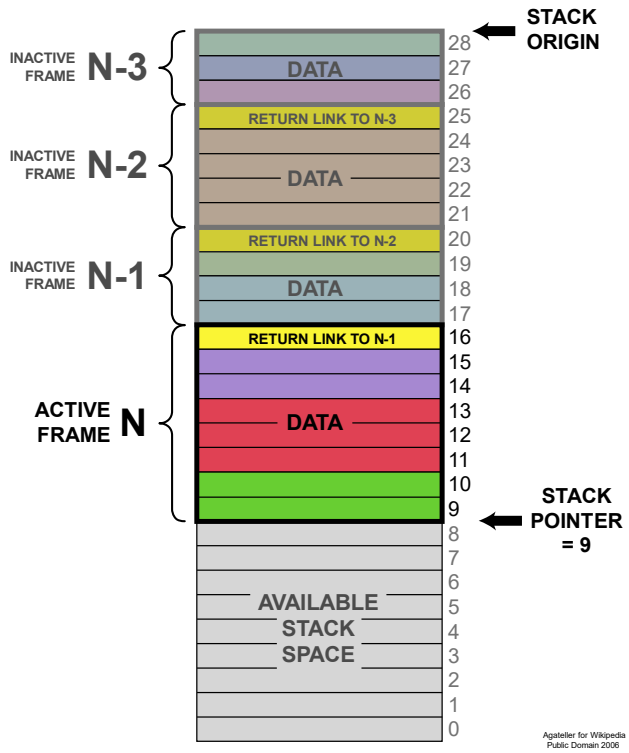


Figure 3.1.5.1 Stack Memory Area

3.1.5.2 Heap Memory Area

Heap is a hierarchical data structure. Access time is slow compared to stack. Heap Space not used efficiently. Memory can be fragmented as frequent blocks of memory are first allocated and then freed. Variables can be resized. Memory is allocated in any random order. Allocation and deallocation is manually done by the programmer. You should use heap only when we require to allocate a large block of memory.

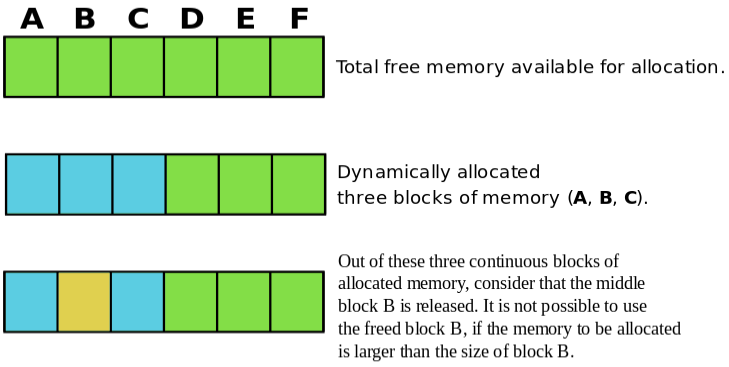


Figure 3.1.5.2 Heap Memory Area

3.1.6 Static Memory Allocation

In static memory allocation - the fixed amount of memory is allocated during the compile time of a program. All the variables in the program are statically allocated. There is no memory reusability. Scope of allocated memory remains from start to end of the program. In this scheme of memory allocation, execution is faster than dynamic memory allocation.

```
void play
{
    int a;
}
int main()
{
    int b;
    int c[10];
    return 1;
}
```

3.1.7 Dynamic Memory Allocation

In dynamic memory allocation - we allocate or deallocate a block of memory during the runtime. The memory is allocated in the heap area. Due to frequent allocation and deallocation the heap can become fragmented. There will be sections of used and unused memory in the allocated space on the heap. C provides the following functions defined under **<stdlib.h>** header file for dynamic memory allocation:

Function	Description
malloc	Allocates the specified number of bytes
realloc	Increases or decreases the size of the pre specified block of memory, moving the block if required
calloc	Allocates the specified number of bytes and initializes them to zero
free	Deallocate the specified block of memory

3.1.7.1 malloc ()

It is used to allocate a memory block in the heap area of the memory of some specified size (in bytes).
Syntax:
(data-type *) malloc(size-in-bytes);
To create an array of integers, is simple:
int arr[50];
The size of array is fixed at compile time. If we wish to allocate a similar array dynamically, following code can be used:
int *arr = malloc(50 * sizeof(int));
This calculates the number of bytes that fifty integers occupy in memory, it requests that many bytes from malloc () and assigns it to a pointer named arr.
If the space is insufficient, memory allocation fails and returns a NULL pointer.
It is good programming practice to check for successful allocation:

```
int *arr = malloc(50 * sizeof(int));
if (arr == NULL) {
 fprintf(stderr, "malloc() failed\r\n");
 return -1;
}
```

3.1.7.2 free ()

When we don't need the dynamic memory, it must call free() to return the memory it occupied to the free the block.
Syntax:
free(arr);

3.1.7.3 calloc ()

It is used to allocate a memory block in the heap area of the memory of some specified size (in bytes). Each block is initialized with a default value '0'.
Syntax:
(data-type*) calloc (n, element-size);
here, n is the no. of elements and element-size is the size of each element.
int *arr = calloc (10, sizeof(int));

3.1.7.4 realloc ()

Using realloc we can resize the amount of memory a pointer points which was previously allocated. If there is pointer acting as an array of size {10} and we want to change array size to {100}, we can use realloc().

```
int *array = malloc(3 * sizeof(int));
array[0] = 1;
array[1] = 2;
```

array[2] = 3;
array = realloc(array, 4 * **sizeof**(int));
array[3] = 4;

Example: Build and run the example code - **test_dynamic_memory_allocation.c**

```
akshay@akshayv:Chapter3_Memory_Management$ gcc test_dynamic_memory_allocation.c -o test_dynamic_memory_allocation
akshay@akshayv:Chapter3_Memory_Management$ ./test_dynamic_memory_allocation
Memory has been successfully allocated by using malloc

 marks = 0x555e732612a0c

 Enter Marks
10
would you like to add more(1/0): 1
Memory has been successfully reallocated using realloc:

 base address of marks are:0x555e732612a0c
 Enter Marks
20
would you like to add more(1/0): 1
Memory has been successfully reallocated using realloc:

 base address of marks are:0x555e732612a0c
 Enter Marks
40
would you like to add more(1/0): 1
Memory has been successfully reallocated using realloc:

 base address of marks are:0x555e732612a0c
 Enter Marks
60
would you like to add more(1/0): 0
marks of students 0 are: 10
 marks of students 1 are: 20
 marks of students 2 are: 40
 marks of students 3 are: 60
akshay@akshayv:Chapter3_Memory_Management$
```

3.1.8 Fragmentation

In fragmentation the storage space is used inefficiently, this reduces the capacity or performance and even both. Fragmentation can lead to storage space being "wasted." When a program starts, there are large and contiguous free memory areas. After some time of use, the long contiguous areas become fragmented into smaller and smaller contiguous regions. After some time with use, it is impossible for program to allocate large contiguous blocks of memory.

Types of Fragmentation:

- Internal Fragmentation
- External Fragmentation

Consider that there is a 10K heap.

Area of 3K is requested:

#define K (1024)

char *ptr1

ptr1 = malloc(3*K);

Then, a further 4K is requested:

ptr2 = malloc(4*K);

Now 3K of memory freed.

After sometime, the first memory allocation, pointed by ptr1, is deallocated:

free(ptr1);

Now we have 6K of memory free in two 3K blocks.

Now a new 4K allocation request is issued:

ptr3 = malloc(4*K);

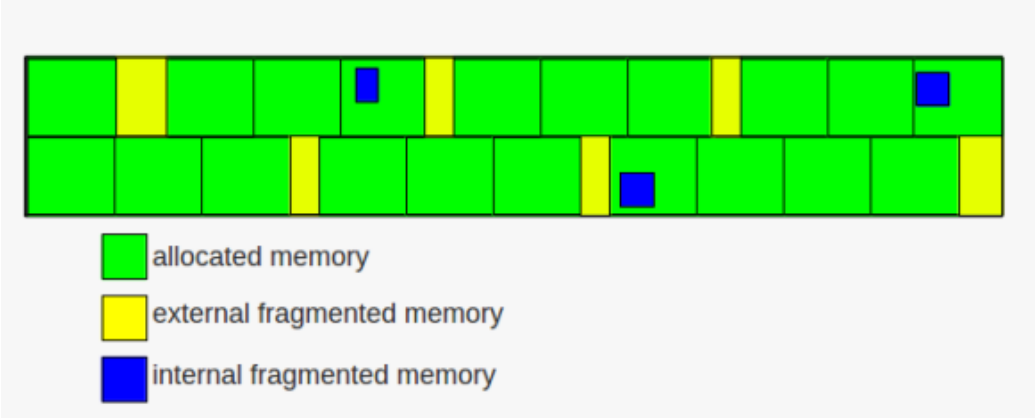
This allocation will fail – NULL pointer is returned to ptr3, even though 6K of memory is available, there is no 4K contiguous block of memory available. This is called memory fragmentation.

Internal Fragmentation:

Internal fragmentation occurs when the memory splits into mounted-sized blocks. Whenever a program request for the memory, the mounted-sized block is allotted to the program. When the memory allotted to the program is larger than the memory requested, then the difference between allotted and requested memory is called internal fragmentation. This is because we have fixed the sizes of the memory blocks. If we use dynamic partitioning for allocating space to the program, we can avoid this problem. In dynamic partitioning, the program is allocated only that amount of space which is required by the program.

External Fragmentation:

In external fragmentation, we have total space available that is needed by a program but still we are not able to allocate the memory because that space is not contiguous. This is because we are allocating memory contiguously. Paging and segmentation (non-contiguous memory allocation) can be used to avoid this problem, memory is allocated non-contiguously.



3.1.9 Memory Leak

Memory leak is when programmers create a memory in the heap area and forget to delete it after use. This reduces the performance of the computer by reducing the amount of available memory. In worst case, all the available memory may be allocated and all or part of the system or device stops working, the application fails, or the system slows down. As a program executes, it needs extra memory and makes an additional request. When an application no longer needs the requested memory or the application closes, it should explicitly release the allocated memory. A programmer should never assume that all the operations are undone when the program exits. We will study tool Val grind to identify memory leaks in the program in upcoming topics.

```c
// C Program to check whether the memory is
// freed or not
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
    int *arr;
    arr = (int *)malloc(sizeof(int));

    if (arr == NULL)
        printf("Memory Is Insuffficient\n");
    else
    {
        free(ptr);
        printf("Memory Freed\n");
    }
}
```

3.1.10 Dangling Pointer

Dangling pointer does not reference a valid object. Dangling pointers emerge during object destruction. When the incoming reference of an object is deleted or deallocated, without modifying the value of the pointer, the pointer will still point to the memory location that was deallocated. The system can reallocate the previously freed memory. Now if the program dereferences the dangling pointer, unpredictable behavior may result, as the memory may contain completely different data. Seconds case if a program writes to memory referenced by a dangling pointer, corruption of data may result. This can lead to subtle bugs that are extremely difficult to find.

Example:

```c
{
    char *dp = NULL;
    /* ... */
    {
        char c;
        dp = &c;
    }
    /* c falls out of scope */
    /* dp is now a dangling pointer */
}
```

We should reset the pointer to null after freeing its reference

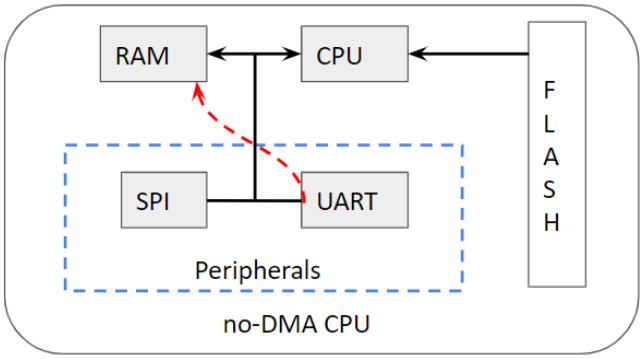
```c
#include <stdlib.h>

void func()
{
    char *dp = malloc(10);
    /* ... */
    free(dp);         /* dp now becomes a dangling pointer */
    dp = NULL;        /* dp is no longer dangling */
    /* ... */
}
```

3.1.11 Direct Memory Access (DMA)

Using Direct Memory Access (DMA) controller we can move of blocks of data without burdening the CPU from:

- Peripheral to memory
- Memory to peripheral
- Memory to memory

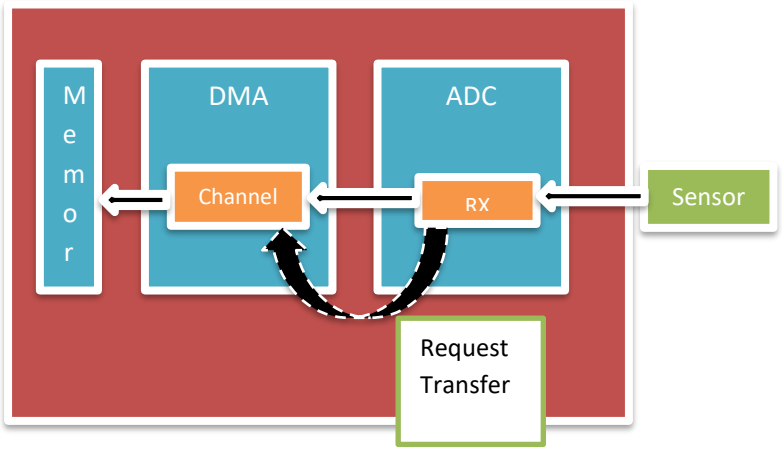
The process is managed by DMA controller (DMAC). In DMA, the CPU first initiates the transfer of data. It does other tasks while the transfer is in progress. Finally, it receives an interrupt from the DMA controller (DMAC) when the transfer is completed. If we have to transfer a data stream of 20kB/s can make a CPU without a DMA be so busy that they miss the timing constraints for the application.



In no-DMA CPU architecture, the CPU does all the work of fetching instructions (code) from flash, executing the decoded instructions, and move data to and from peripherals and memory.
For eg: UART data receiver that gets a stream of data that the CPU has to transfer to a local buffer so as not to lose any data packets.
This generates a large number of interrupts per second being fired by different peripherals (ADC, UART, SPI, etc.).
The CPU has to shuffle everything and lose more and more time by switching between these interrupts and main tasks.

On an event, the peripheral sends a request signal to the DMA Controller. The DMA controller serves the request depending on the channel priorities. When the DMA Controller accesses the peripheral, an Ack is sent to the peripheral by the DMA Controller. The peripheral releases its request when it gets the Ack from the DMA Controller. When the request is de-asserted by the peripheral, the DMA Controller releases the Ack. For more requests, the peripheral can initiate the next transaction. Here are a handful of possible scenarios when DMA can be used:

- UART data reception from a serial device to a local buffer.
- ADC circular buffer conversions when we have multiple channels with sensors connected.
- SPI external flash memory interfacing for high-speed data logging.
- SPI camera, high speed ADC interfacing.
- Local buffer to local buffer data transfer.

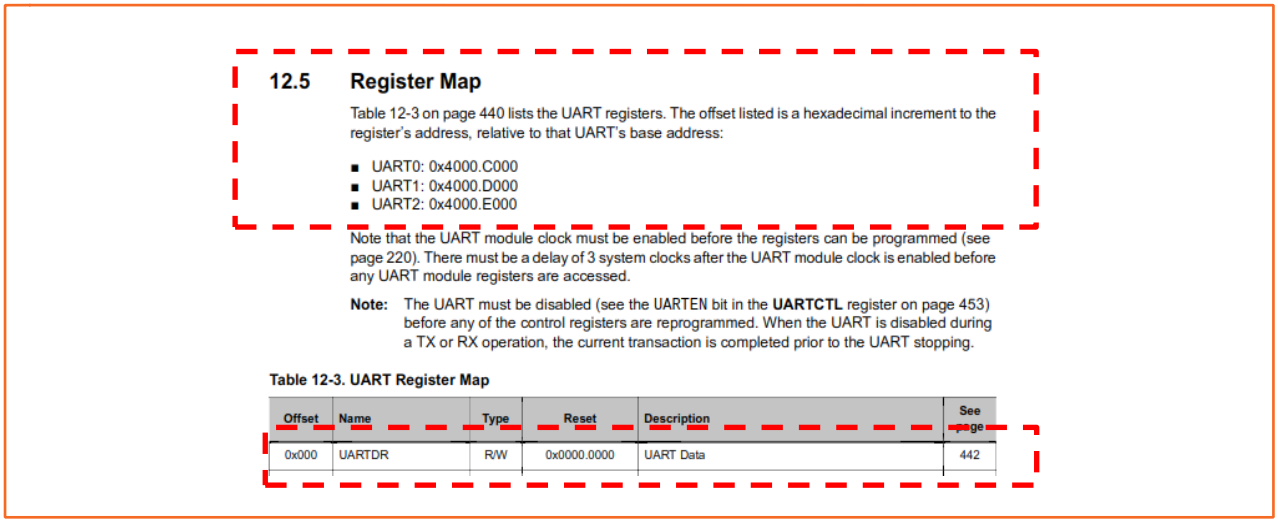


With the use of the DMA unit, we can direct the data from the ADC peripheral directly to the memory while the CPU doing other tasks.

3.1.12 Memory Mapped Input Output (MMIO)

The CPU cannot do anything by itself. CPU needs to be linked with peripherals, memory, or IO devices. This linking is called Interfacing. Memory-mapped I/O (MMIO) and Port-mapped I/O (PMIO) are two methods of performing input/output (I/O) between the CPU and peripheral devices in a computer. Memory-mapped I/O uses the same address space to address both I/O devices and main memory. The registers of the I/O devices and memory is mapped to address values. Each device monitors the CPU's address bus & responds to any CPU access of an address assigned to that device by connecting the data bus to the desired device's hardware register. Consider example below: When we want to send data over UART in bare metal application we can directly write data to UART data register at the address of register i.e. : **PERIPHERAL_BASE_ADDRESS + REG_OFSET.**

```
// Uart0 Peripheral Base address: 0x4000C000
volatile unsigned int * lm3s6965_uart0 = (unsigned int*)0x4000C000;
// Function to send data to uart0
void uart0_print(const char* msg)
{
    while(*msg)
    {
        // we can talk directly to the hardware by
        // writing to a set of predetermined memory addresses
        // UARTDR => Uart Data Register offset 0x0 from Base Peripheral address:
        *lm3s6965_uart0 = *msg;
        msg++;
    }
}
```



12.5 Register Map

Table 12-3 on page 440 lists the UART registers. The offset listed is a hexadecimal increment to the register's address, relative to that UART's base address:

- UART0: 0x4000.C000
- UART1: 0x4000.D000
- UART2: 0x4000.E000

Note that the UART module clock must be enabled before the registers can be programmed (see page 220). There must be a delay of 3 system clocks after the UART module clock is enabled before any UART module registers are accessed.

Note: The UART must be disabled (see the UARTEN bit in the **UARTCTL** register on page 453) before any of the control registers are reprogrammed. When the UART is disabled during a TX or RX operation, the current transaction is completed prior to the UART stopping.

Table 12-3. UART Register Map

Offset	Name	Type	Reset	Description	See page
0x000	UARTDR	R/W	0x0000.0000	UART Data	442

3.1.13 Static Code Analysis

Static program analysis of a computer program is done without executing them. The analysis is performed by an automated tool, with human analysis typically being called program comprehension, program understanding or code review. The automated tool will scan all source code to check for vulnerabilities while validating the code. The analysis is performed on some version of a source code, and, in some cases, on some form of object code. This helps developers to understand the code base and ensure it is compliant, safe, and secure.

Static analysis is generally used in finding coding issues such as:

- Security vulnerabilities
- Coding standard violations, e.g., C Coding Standard - SEI CERT
- Undefined values
- Programming errors
- Syntax violations

The static analysis is useful for addressing vulnerabilities in application code, for e.g., buffer overflows.

How is static analysis done:

When code is written, a static code analyzer should be used to look over the code. It will check against characterized coding rules from standards or custom predefined rules. When the code is gone through the static code analyzer, the analyzer will have recognized whether the code follows the set guidelines. In some cases, it's possible for the tool to flag false positives, so somebody should go through and excuse any. When false positives are deferred, developers can begin to fix any obvious errors, starting with the most critical ones. When the code issues are settled, the code can continue on towards testing through execution.

Types of static analysis:

Control analysis - centers around the control flow in a calling structure. For instance, a control flow could be a method, process, function, or in a subroutine.

Data analysis - ensures characterized data is appropriately utilized while also making sure data objects are appropriately operating.

Fault and failure analysis - analyzes faults and failures.

Interface analysis - confirms simulations to check the code and ensures the interface fits into the model and simulation.

Benefits of Static Analysis:

Implementation

Static analysis can be put into practice early in the software development lifecycle (SDLC), it will give more time to fix the errors discovered by the tool.

Static analysis can detect the exact line of code that's been found to be vulnerable.

Security

Security is major concern and by adopting static analysis tools you can cut uncertainty of security vulnerabilities in your application, which will ensure that you are delivering a secure and reliable software.

Automation

Automation can save time and energy which means you can invest efforts in some other aspects of development lifecycle, this ensures faster software release.

Define rules to assist developers

Developers work on projects without knowledge about security while they code. Static code analyzers help you characterize project specific rules. This ensures all developers follow them without any manual intervention or sidetracking.

Tools for Static Analysis:

- PC-lint Plus - Support for coding Standards such as MISRA, AUTOSAR, and CERT C
- SonarQube
- Sparse - tool designed to find possible coding faults in the Linux kernel.
- Lint
- Splint - An open-source tool statically analyzes C programs for coding mistakes & security vulnerabilities.
- Cpplint

And the list goes on you can select the tools based on variety of factors like industrial standards supported, license/ Opensource, Programming Language supported etc.

3.1.13.1 Static Code Analysis - Splint

SPLINT - Secure Programming Lint:

Splint is a tool for statically analyzing C programs for coding mistakes and security vulnerabilities. With minimal effort, Splint can be used as a better lint. If some efforts are invested, adding annotations to programs, Splint can perform strong checking than that can be done by any standard lint. Check manual pages (man splint) for complete details of the splint.

Install the SPlint using: sudo apt-get install splint

To do static analysys run the command:

splint <filename.c>

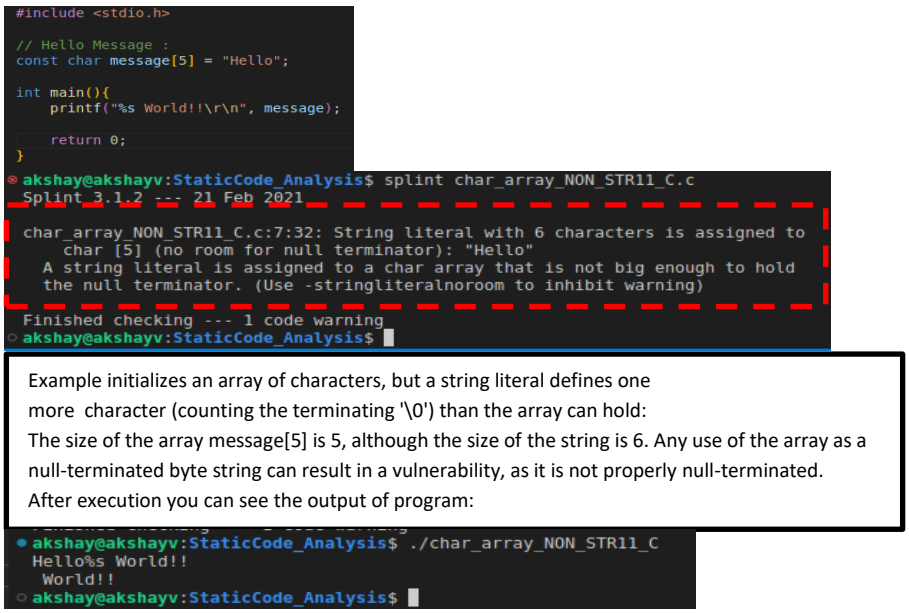
Examples for static analysis using SPlint:

Program: char_array_NON_STR11_C.c -

Example code not compliant with STR11-C.

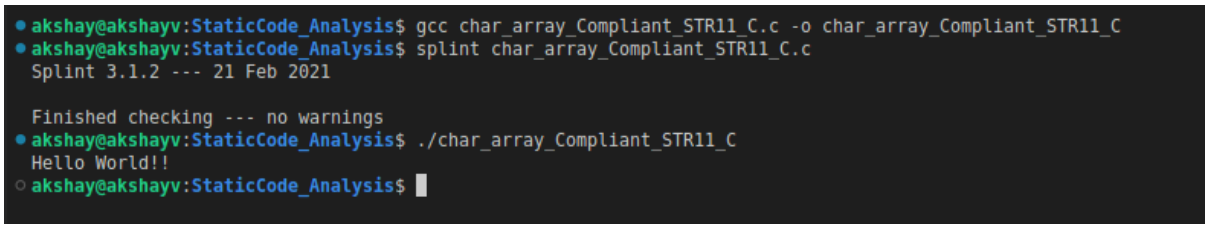
Run static analysis with:

splint char_array_NON_STR11_C.c



Example initializes an array of characters, but a string literal defines one more character (counting the terminating '\0') than the array can hold:
The size of the array message[5] is 5, although the size of the string is 6. Any use of the array as a null-terminated byte string can result in a vulnerability, as it is not properly null-terminated.
After execution you can see the output of program:

After addressing the issues pointed by Splint. Consider the program: **char_array_Compliant_STR11_C.c** Splint shows no warnings and output of program is also correct.



Consider the example: **buffer_overflow_example.c**. The program accepts the password from user. Imagine a scenario where the assailant comes to know about a buffer over flow in program and he/she takes advantage of it. What if it is undetected & this code is deployed in ATM machine? Very bad consciences can be faced. Splint analysis below shows 4 warnings.

3.1.14 Memory Profiling

Run-time memory errors and leaks are the most difficult errors to locate and the most important to correct. These symptoms of incorrect memory use are unpredictable and typically appeared far from the root of the error. These errors often remain undetected until they are triggered by a random event. The program can appear to work correctly when, in fact, it's only working by chance. Memory Profiling analyzes memory usage and detects memory leaks. In this module, we introduce Val grind tool: Memcheck.

Valgrind:

Valgrind is a tool for profiling and debugging Linux programs. It is useful to automatically detect memory management & threading errors, making programs more stable & robust.

It consists of a core, that provides a synthetic CPU in software & a series of debugging and profiling tools.

Memcheck tool

Memcheck tool adds extra instrumentation code around most instructions, which keeps track of the addressability and validity.

Memcheck changes the standard C memory allocator with its own implementation, this also includes memory guards around all allocated blocks.

The Memcheck may detect and warn the following issues:

- Potential memory leaks
- Use of uninitialized memory
- Reading or writing in illegal sections
- Reading or writing memory after it has been released

3.1.15 Valgrind - Memcheck

How to use Memcheck tool:

Compile program with the flag -g, it generates debug information, we can see line numbers in the output. To test the program with valgrind memcheck tool:

Use command: **valgrind --tool=memcheck <program name>**

Consider simple program with no errors: **hello_good.c**

Compile code with:

gcc -Wall -g hello_good.c -o hello_good

To use valgrind:

valgrind --tool=memcheck ./hello_good

This example does not have any error Valgrind can detect with Memcheck.

Thus Valgrind-Memcheck, reported zero errors.

```c
#include <stdlib.h>
int main()
{
  return 0;
}
```

```
akshay@akshayv:MemoryProfiling$ valgrind --tool=memcheck  ./hello_good
==60326== Memcheck, a memory error detector
==60326== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==60326== Using Valgrind-3.18.1 and LibVEX; rerun with -h for copyright info
==60326== Command: ./hello_good
==60326==
==60326==
==60326== HEAP SUMMARY:
==60326==     in use at exit: 0 bytes in 0 blocks
==60326==   total heap usage: 0 allocs, 0 frees, 0 bytes allocated
==60326==
==60326== All heap blocks were freed -- no leaks are possible
==60326==
==60326== For lists of detected and suppressed errors, rerun with: -s
==60326== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
akshay@akshayv:MemoryProfiling$
```

3.1.15.1 Memcheck Leak in Code:

Consider the program: **hello_memory_leak.c**. We introduce memory leak into the code with a **malloc** in the main code. We will allocate **10 bytes** of memory to **ptr** using malloc. We didn't free the allocated memory.

Compile the code:

gcc -Wall -g hello_memory_leak.c -o hello_memory_leak

Test with valgrind

valgrind --tool=memcheck --leak-check=full ./hello_memory_leak

```
akshay@akshayv:MemoryProfiling$ splint hello_memory_leak.c
Splint 3.1.2 --- 21 Feb 2021

hello_memory_leak.c: (in function main)
hello_memory_leak.c:8:4: Dereference of possibly null pointer ptr: *ptr
  A possibly null pointer is dereferenced.  Value is either the result of a
  function which may return null (in which case, code should check it is not
  null), or a global, parameter or structure field declared with the null
  qualifier. (Use -nullderef to inhibit warning)
   hello_memory_leak.c:7:13: Storage ptr may become null
hello_memory_leak.c:9:12: Fresh storage ptr not released before return
  A memory leak has been detected. Storage allocated locally is not released
  before the last reference to it is lost. (Use -mustfreefresh to inhibit
  warning)
   hello_memory_leak.c:7:24: Fresh storage ptr created

Finished checking --- 2 code warnings
akshay@akshayv:MemoryProfiling$
```

Note: We can also use Static analysis to detect memory leak with **Splint tool**

```
4   #include <stdlib.h>
5   int main()
6   {
7     char* ptr=malloc(10);
8     *ptr = 'a';
9     return 0;
10  }
```

```
akshay@akshayv:MemoryProfiling$ gcc -Wall -g hello_memory_leak.c -o hello_memory_leak
akshay@akshayv:MemoryProfiling$ valgrind --tool=memcheck  --leak-check=full  ./hello_memory_leak
==61104== Memcheck, a memory error detector
==61104== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==61104== Using Valgrind-3.18.1 and LibVEX; rerun with -h for copyright info
==61104== Command: ./hello_memory_leak
==61104==
==61104==
==61104== HEAP SUMMARY:
==61104==     in use at exit: 10 bytes in 1 blocks
==61104==   total heap usage: 1 allocs, 0 frees, 10 bytes allocated
==61104==
==61104== 10 bytes in 1 blocks are definitely lost in loss record 1 of 1
==61104==    at 0x4848899: malloc (in /usr/libexec/valgrind/vgpreload_memcheck-amd64-linux.so)
==61104==    by 0x10915E: main (hello_memory_leak.c:7)
==61104==
==61104== LEAK SUMMARY:
==61104==    definitely lost: 10 bytes in 1 blocks
==61104==    indirectly lost: 0 bytes in 0 blocks
==61104==      possibly lost: 0 bytes in 0 blocks
==61104==    still reachable: 0 bytes in 0 blocks
==61104==         suppressed: 0 bytes in 0 blocks
==61104==
==61104== For lists of detected and suppressed errors, rerun with: -s
==61104== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 0 from 0)
akshay@akshayv:MemoryProfiling$
```

3.1.15.2 Memcheck Leak in Code Fix

Memory Leak in Code Fix:

When we use malloc for dynamic memory allocation. We should take care that we free the allocated memory before exiting the block/program.

Consider the program: **hello_memory_leak_fix.c.**

Compile with:

gcc -g hello_memory_leak_fix.c -o hello_memory_leak_fix

To use valgrind:

valgrind --tool=memcheck --leak-check=full ./hello_memory_leak_fix

Valgrind output also shows no errors and all the allocated memory is freed.

```
4   #include <stdlib.h>
5   int main()
6   {
7      void* ptr=malloc(10);
8      free(ptr);
9      return 0;
10  }
```

```
akshay@akshayv:MemoryProfiling$ gcc -g hello_memory_leak_fix.c -o hello_memory_leak_fix
akshay@akshayv:MemoryProfiling$ valgrind --tool=memcheck  --leak-check=full  ./hello_memory_leak_fix
==62316== Memcheck, a memory error detector
==62316== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==62316== Using Valgrind-3.18.1 and LibVEX; rerun with -h for copyright info
==62316== Command: ./hello_memory_leak_fix
==62316==
==62316==
==62316== HEAP SUMMARY:
==62316==     in use at exit: 0 bytes in 0 blocks
==62316==   total heap usage: 1 allocs, 1 frees, 10 bytes allocated
==62316==
==62316== All heap blocks were freed -- no leaks are possible
==62316==
==62316== For lists of detected and suppressed errors, rerun with: -s
==62316== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
akshay@akshayv:MemoryProfiling$
```

3.1.15.3 Illegal Read/Write

One of the problems detected is warning about the code that it is reading or writing illegal positions from a program. The example consists of an illegal writing on the zero-memory address.

Illegal access is produced by writing a memory address that is not part of the program and releasing the alarm Memcheck:

Consider the program: **illegal_write.c**

Valgrind reports an **Invalid write of size 4** for this issue in the main code.

```
3   #include <stdlib.h>
4   int main()
5   {
6     // Pointing Zero memory addr
7     int *ptr=0;
8     // Writing to Zero memory addr
9     // Note: address is not in program
10    (*ptr)=33;
11    return 0;
12  }
```

```
akshay@akshayv:MemoryProfiling$ gcc -g illegal_write.c -o illegal_write
akshay@akshayv:MemoryProfiling$ valgrind  ./illegal_write
==62841== Memcheck, a memory error detector
==62841== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==62841== Using Valgrind-3.18.1 and LibVEX; rerun with -h for copyright info
==62841== Command: ./illegal_write
==62841==
==62841== Invalid write of size 4
==62841==    at 0x10913D: main (illegal_write.c:10)
==62841==  Address 0x0 is not stack'd, malloc'd or (recently) free'd
==62841==
==62841==
==62841== Process terminating with default action of signal 11 (SIGSEGV)
==62841==  Access not within mapped region at address 0x0
==62841==    at 0x10913D: main (illegal_write.c:10)
==62841==  If you believe this happened as a result of a stack
==62841==  overflow in your program's main thread (unlikely but
==62841==  possible), you can try to increase the size of the
==62841==  main thread stack using the --main-stacksize= flag.
==62841==  The main thread stack size used in this run was 8388608.
==62841==
==62841== HEAP SUMMARY:
==62841==     in use at exit: 0 bytes in 0 blocks
==62841==   total heap usage: 0 allocs, 0 frees, 0 bytes allocated
==62841==
==62841== All heap blocks were freed -- no leaks are possible
==62841==
==62841== For lists of detected and suppressed errors, rerun with: -s
==62841== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 0 from 0)
Segmentation fault (core dumped)
akshay@akshayv:MemoryProfiling$
```

3.1.15.4 Uninitialized Variables

Valgrind can detect issues caused by uninitialised variables.

For example:

Consider program: uninitialised_variable.c

Compile:

gcc -g uninitialised_variable.c -o uninitialised_variable

Test with Valgrind:

valgrind --track-origins=yes./uninitialised_variable

Valgrind shows the following error in the output:

Similary there are may use issues that valgrind can report

To read more about valgrind read Valgrind(1) - Linux man

```c
#include <stdlib.h>
#include <stdio.h>
int main()
{
    // Should be initialized (i.e. to zero).
    int number;
    if (number == 0)
    {
        printf("number is zero");
    }

    return 0;
}

```

```
akshay@akshayv:MemoryProfiling$ gcc -g uninitialised_variable.c -o uninitialised_variable
akshay@akshayv:MemoryProfiling$ valgrind --track-origins=yes ./uninitialised_variable
==63879== Memcheck, a memory error detector
==63879== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==63879== Using Valgrind-3.18.1 and LibVEX; rerun with -h for copyright info
==63879== Command: ./uninitialised_variable
==63879==
==63879== Conditional jump or move depends on uninitialised value(s)
==63879==    at 0x109159: main (uninitialised_variable.c:9)
==63879==  Uninitialised value was created by a stack allocation
==63879==    at 0x109149: main (uninitialised_variable.c:6)
==63879==
number is zero==63879==
==63879== HEAP SUMMARY:
==63879==     in use at exit: 0 bytes in 0 blocks
==63879==   total heap usage: 1 allocs, 1 frees, 1,024 bytes allocated
==63879==
==63879== All heap blocks were freed -- no leaks are possible
==63879==
==63879== For lists of detected and suppressed errors, rerun with: -s
==63879== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 0 from 0)
akshay@akshayv:MemoryProfiling$
```

4. Advanced Data Types

Unit 4.1 - Variable Length Data Structures

Unit 4.2 - Structures and Unions

Key Learning Outcomes

At the end of this module, you will be able to:

1. Apply the validate length data structures in the code.
2. Apply the structures and unions in the code.

UNIT 4.1: SDLC - Security Development Life Cycle

Unit Objectives

At the end of this unit, you will be able to:

1. Apply the user defined data type "typedef" in the code.
2. Apply the user "typecasting" in the code.
3. Apply the user variable and flexible length array in the code.

4.1.1 Typedef

Keyword **typedef** can be used to give a type a new name. It is used to provide meaningful names to already existing variables in the program.

Syntax: **typedef <existing name> <alias name>**

Existing name: name of an existing variable.

Alias name: name given to the existing variable by a developer.

Example:

typedef char BYTE;

Once an alias name 'BYTE' is given to a pre-defined term 'char', the identifier 'BYTE' can be used as an abbreviation for the same.

BYTE b1, b2;

This will declare the variables **b1** & **b2** of char data type.

4.1.2 Typecast

Converting the data type from one form to another is known as type casting or the type conversion. Programmer can initiate a data type of **int** and make it work like another data type like **char** or **float** for one single operation.

Syntax: **(type name) expression**

Types of typecasting:

Implicit Type casting:

Type conversion is performed automatically by the compiler you are working on without the programmer's intervention.

```
int x = 10;
printf ("%c", x); // it is implicitly converted from int to char
```

Explicit Type casting:

Type of conversion that happens due to human intervention.

A developer will perform it by posing the type of the variable.

```
int x = 10;
printf ("%c", (char)x); /* it is explicitly converted from int to char
```

Rules to be followed while doing a type conversion:

All the character types must be converted to integer types.

All the integer types to be converted to float.

All the float types to be converted to a double.

4.1.3 Structure

Structure stores the different types of elements. A structure can contain many different data types (int, float, char, nested structures). Each variable in the structure is known as a **member** of the structure. The struct keyword is used to define structure.

Syntax:

struct structure_name
{
data_type member1;
data_type memeberN;
};

- Declaration of Structure variables:

struct structure_name
{
data_type member1;
data_type memeberN;
}struct_var1,struct_var2**;**

Or

struct structure_name struct_var1, struct_var2;

```c
// Create a structure called car
struct Car
{
    char brand[50]; // string (char array) Member
    char model[50]; // string (char array) member
    int year;       // int variable
    float milage;   // float variable
} car1,car4;        // variable decleration along with structure
```

4.1.3.1 Pointers to Structure

A structure pointer is defined as the pointer which points to the address of the memory block that stores a structure. Complex data structures like Linked lists, trees, graphs, etc. are created with the help of structure pointers. The structure pointer tells the address of a structure in memory by pointing the variable to the structure variable.

The members of struct can be accessed in two ways:

- With the help of (*) asterisk or indirection operator and (.) dot operator.
- With the help of (->) Arrow operator.

```c
struct marks {
   int math;
   int history;
}

struct marks Roll1 = { 83, 75 };
/* p is a pointer to Roll1 */
struct marks *p = &Roll1;
/* set the first member of the struct */
(*p).math = 89;
/* equivalent method to set the first member of the struct */
p->math = 89;
```

Example Program: Usage of Structure Ptr

test_structure_ptr.c

```
akshay@akshayv:Chapter5_Referencing_data_and_functions$ gcc test_structure_ptr.c -o test_structure_ptr
akshay@akshayv:Chapter5_Referencing_data_and_functions$ ./test_structure_ptr
 Display The Data:
 Roll Number: 34
 Name: Akshay Varpe
 Branch: Electronics & Telecommunication
 Batch: 2017
akshay@akshayv:Chapter5_Referencing_data_and_functions$
```

Nested Structures:

A structure inside another structure is known as nested structure.

For example:

We can access the items using **(.) dot** operator.

To get milage of structure variable RegCar:

float milageCar1 = RegCar.VehicleInfo.milage;

```c
// Create a structure called car
struct Car
{
    char brand[50]; // string (char array) Member
    char model[50]; // string (char array) member
    int year;       // int variable
    float milage;   // float variable
} car1,car4;        // variable decleration along with structure

// Create a Nested Structure called RegDetails
struct RegDetails
{
    int VehilceID;
    char RegNum[12];
    struct Car VehicleInfo; // struct Car as a member of RegDetails
} RegCar;   // Variable for RegDetails
```

Array of Structures:

We can define array of structures using syntax:

struct < struct name > <array name>[array size];

```
// Create a structure called car
struct Car
{
    char brand[50]; // string (char array) Member
    char model[50]; // string (char array) member
    int year;       // int variable
    float milage;   // float variable
} car1,car4;        // variable decleration along with structure
```

For example:

Consider the above structure, to declare the array we use:

struct Car CarList[5]; // Array of 5 Structures

To assign values:

CarList[0] = {"BMW", "X5", 1999, 15.3};

CarList[1] = {"Ford", "Mustang", 1969, 10.6};

To get milage vale from structure CarList[1]

float milageCar1 = CarList[1].milage;

Copying Structures:

Structure is treated like a value which occupies given number of bytes. To copy structure along with all members we can use simple assignment operator

For example: Consider structure Car

struct Car carCopy, car1;

car1 = {"BMW", "X5", 1999, 15.3};

To Copy car1 to carCopy structure:

carCopy = car1;

To copy using pointer:

struct Car *carPtr1, *carPtr2;

carPtr1 = & car1;

carPtr2 = & carCopy;

*carPtr2 = *carPtr1;

```
// Create a structure called car
struct Car
{
    char brand[50]; // string (char array) Member
    char model[50]; // string (char array) member
    int year;       // int variable
    float milage;   // float variable
} car1,car4;        // variable decleration along with structure
```

Passing structure to function:

We can pass structure to function as a value or as a reference. Consider the following functions that have structures as parameters.

struct Car car1;

struct Car car2 = {"BMW", "X5", 1999, 15.3};

FuncFillCarDetaials(car2, &car1);

FuncDisplayCar(car1);

```
// Create a structure called car
struct Car
{
    char brand[50]; // string (char array) Member
    char model[50]; // string (char array) member
    int year;       // int variable
    float milage;   // float variable
} car1,car4;        // variable decleration along with structure
```

```
// Function with Structure as argument
// Function will print the values of members from the structures
void FuncDisplayCar(struct Car s)
{
    printf("Brand=%s\tModel=%s\tYear=%d\tMilage=%f\r\n", s.brand, s.model, s.year, s.milage);
}

// Passing struct by reference
// This function will assign the values to the structure passed by reff
void FuncFillCarDetaials(struct Car details, struct Car * s)
{
    s->year = details.year; // Assign the value to sruct argument passed by reference
    s->milage = details.milage;
    strcpy(s->brand, details.brand);
    strcpy(s->model, details.model);
}
```

Example program: **test_structure.c** shows how to use the structure. Note: Read the comments in the code carefully. The following operations are carried out by the code:

- Assign value to structure.
- How to use string as structure member.
- Structure variable declaration
- Copy Structure
- Structure array operations
- Pass structure to function.
- Pass structure to function by reference.
- Return the structure from function.
- Nested structure

```
akshay@akshayv:Chapter4_Advanced_Data_Types$ gcc test_structure.c -Wall -o test_structure
akshay@akshayv:Chapter4_Advanced_Data_Types$ ./test_structure
Print all the assigned values:
Brand=Toyota    Model=Corolla   Year=2011       Milage=21.500000
Brand=BMW       Model=X5        Year=1999       Milage=15.300000
Brand=Ford      Model=Mustang   Year=1969       Milage=10.600000
Brand=Maruti    Model=Swift     Year=2019       Milage=28.500000
Brand=Toyota    Model=Corolla   Year=2011       Milage=21.500000

carCopy:
Brand=Toyota    Model=Corolla   Year=2011       Milage=21.500000

Print the CarList:
CarList[0]:
Brand=Toyota    Model=Corolla   Year=2011       Milage=21.500000
CarList[1]:
Brand=BMW       Model=X5        Year=1999       Milage=15.300000
CarList[2]:
Brand=Ford      Model=Mustang   Year=1969       Milage=10.600000
CarList[3]:
Brand=Maruti    Model=Swift     Year=2019       Milage=28.500000
CarList[4]:
Brand=Toyota    Model=Corolla   Year=2011       Milage=21.500000

Nested Struct:
RegCar.VehilceID = 12
RegCar.RegNum    = MH12LX4122
Brand=BMW       Model=X5        Year=1999       Milage=15.300000
akshay@akshayv:Chapter4_Advanced_Data_Types$
```

4.1.4 Structure Padding

Structure members are assigned to memory addresses in incremental order. The first member starts at the beginning address of the structure name itself. Structure padding is a concept in C that adds one or more empty bytes between the memory addresses to align the data in memory. The processor does not read 1 byte at a time. It reads 1 word at a time.

What does the 1 word mean?

If we have a **32-bit processor**, then the processor reads 4 bytes at a time **1 word = 4 bytes.**

If we have a **64-bit processor**, then the processor reads 8 bytes at a time, **1 word = 8 bytes.**

- As we know that structure occupies the contiguous block of memory as shown in diagram. Consider the **32-bit architecture:**

The problem is that in one CPU cycle we can access:

- one byte of **char a**, one byte of **char b**, bytes of **int c**

We will not face any problem while accessing the **char a** and **char b** as both the variables can be accessed in **one CPU cycle**. We will face the problem when we access the **int c** variable as **2 CPU cycles** are required to access the value of the 'c' variable.

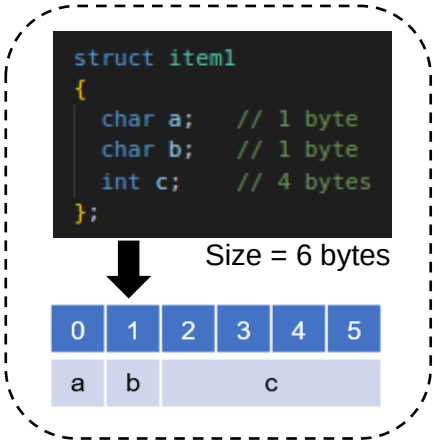


Fig 4.1.4 CPU Cycle

Suppose we only want to access the variable 'c', which requires two cycles. The variable 'c' is of 4 bytes, so it can be accessed in one cycle also, but in this scenario, it is utilizing 2 cycles. This is an unnecessary wastage of CPU cycles. Due to this reason, the structure padding concept was introduced to save the number of CPU cycles.

How is structure padding done?

To achieve the structure padding, an empty row is created on the left, as shown in the diagram. The two bytes which are occupied by the 'c' variable on the left are shifted to the right. So, all four bytes of 'c' variable are on the right. Now, the 'c' variable can be accessed in a single CPU cycle. After structure padding, the total memory occupied by the structure is 8 bytes (1 byte +1 byte +2 bytes +4 bytes). Although the memory is wasted in this case, the variable can be accessed within a single cycle.

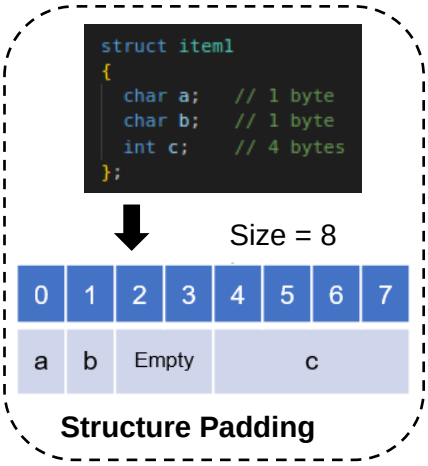


Fig 4.1.4 Structure Padding

The structure padding is an in-built process that is automatically done by the compiler. Sometimes it required to avoid the structure padding in C as it makes the size of the structure greater than the size of the structure members.

Structure Packing:

Structure Packing, prevents compiler from doing padding means remove the unallocated space allocated by structure. We can avoid the structure padding in C in two ways:

- Using #pragma pack (1) directive
- Using __attribute__((__packed__))
- Example Program: Demo for strcut padding: **test_structure_padding.c**

```c
// To avoid the structure padding.
// Use the #pragma pack(1) directive
#pragma pack(1)
struct item3
{
    char a;  // 1 byte
    char b;  // 1 byte
    int  c;  // 4 byte
};

// By using attribute
struct item4
{
    char a;  // 1 byte
    char b;  // 1 byte
    int  c;  // 4 bytes
}__attribute__((packed));;
```

```
akshay@akshayv:Chapter4_Advanced_Data_Types$ gcc test_structure_padding.c -o test_structure_padding
akshay@akshayv:Chapter4_Advanced_Data_Types$ ./test_structure_padding
The size of the item1 structure = 8
The size of the item2 structure = 12
The size of the item3 structure = 6
The size of the item4 structure = 6
akshay@akshayv:Chapter4_Advanced_Data_Types$
```

4.1.5 Union

Union is a user-defined data type in C, which stores a collection of different kinds of data, just like a structure. To access any member of a union, we use the **member access operator (.)**. With unions, only one member can contain a value at any given time. Unions provide an efficient way of using the same memory location for multiple-purpose. The memory occupied by a union is large enough to hold the largest member of the union. Consider the example union data:

Size of union = 20, as the max space is occupied by the char str[20] i.e. 20bytes.

Syntax:

```
union unionName
{
   data_type member1;
   data_type memeberN;
};
```

Example:

```
union Data {
  int i;              // 1 byte
  float f;           // 4 bytes
  char str[20];  // 20 bytes
} data;
```

Example Program: **test_union.c** shows how to use the union. Assign values to all member at once:

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

Use one variable at a time:

Here, all the members are getting printed very well because one member is being used at a time.

```
akshay@akshayv:Chapter4_Advanced_Data_Types$ gcc test_union.c -o test_union
akshay@akshayv:Chapter4_Advanced_Data_Types$ ./test_union

Assign values to all members at a time:
data.i : 1953724755
data.f : 771599660719905792110735162081280.000000
data.str : System Programing

Now we will use one variable at a time:
data2.i : 10
data2.f : 220.500000
data2.str : System Programing
akshay@akshayv:Chapter4_Advanced_Data_Types$
```

4.1.6 Struct vs Union

Struct	Union
• Complier allocates memory to each member separately. Size of struct is equal to the sum of sizes of its members. • Each member is assigned a unique storage area. • Altering the value of a member will not affect other members of the structure. • Individual members can be accessed at a time. • Several members of the structure can initialize at once.	• Compiler allocates the memory considering the size of largest member in union. Size of union is equal to size of largest member in union. • Memory allocated is shared by all the members. • Altering the value of member will affect other member values. • Only one member can be accessed at a time. • Only one member of a union can be initialized.

4.1.7 Processor Endianness

Endianness is a term that describes the order in which a sequence of bytes is stored in computer memory.

Types of Endianness:

- **Little endian:**

Last byte of binary representation of the multibyte data-type is stored first. **X86** processor and most of the **ARM Cortex-M3** based microcontrollers use the little-endian format.

- **Big endian:**

First byte of binary representation of the multibyte data-type is stored first. **Motorola 6800 / 6801** processors use the big-endian format.

Example:

Integer is stored as 4bytes. will be stored as following:

int x = 0x01234567;

Big Endian	
Address	Data
0x100	0x01
0x101	0x23
0x102	0x45
0x103	0x67

Little Endian	
Address	Data
0x100	0x67
0x101	0x45
0x102	0x23
0x103	0x01

Fig 4.1.7 Types of Endianness

Most of the times compiler takes care of endianness. When we perform a bit-wise operation on an integer then compiler automatically handles the endianness. However, endianness becomes an issue in following cases:

Writing raw bytes of data to a file in one processor and you send it to a system that uses different endian processor. Send bytes of data over network as a serialized steam of data from one endian processor system to other endian processor system. In network communication, TCP/IP suites are defined to be big-endian. Sometimes it matters when you are using type casting, below program is an example.

Consider the example in image:

A char array is typecasted to an integer type.

On little endian machine we get:

DataInt = 1

On Big endian machine we get:

DataInt = 16777216

Example Program: To find Endianness of system

test_endianness.c

```
akshay@akshayv:Chapter4_Advanced_Data_Types$ gcc test_endianness.c -o test_endianness
akshay@akshayv:Chapter4_Advanced_Data_Types$ ./test_endianness
int Data = 0x01234567

    Address      Data
0x7ffd6d53beb4   0x67
0x7ffd6d53beb5   0x45
0x7ffd6d53beb6   0x23
0x7ffd6d53beb7   0x01
```

Example Program:

test_endianness_issue.c

```c
#include <stdio.h>
int main()
{
    unsigned char DataArr[4] = {0x01, 0x00, 0x00, 0x00 };
    // char array is typecasted to an integer type.
    int DataInt = *(int *) DataArr;
    printf("DataInt = %d\r\n", DataInt);
    return 0;
}
```

```
akshay@akshayv:Chapter4_Advanced_Data_Types$ gcc test_endianness_issue.c -o test_endianness_issue
akshay@akshayv:Chapter4_Advanced_Data_Types$ ./test_endianness_issue
DataInt = 1
akshay@akshayv:Chapter4_Advanced_Data_Types$
```

Sometimes you may need to convert the endianness of data. Using the union, we change the endianness of data:

Example program:

test_change_endianness_union.c

```
akshay@akshayv:Chapter4_Advanced_Data_Types$ gcc test_change_endianness_union.c -o test_change_endianness_union
akshay@akshayv:Chapter4_Advanced_Data_Types$ ./test_change_endianness_union
Orignal Byte Order    = 0x11223344
Converted Byte Order = 0x44332211
akshay@akshayv:Chapter4_Advanced_Data_Types$
```

Network Byte Order:

Network stacks and communication have a defined endianness, to allow two nodes with different endianness to communicate. In network communication, **TCP/IP suites are big-endian.** Any 16-bit/32-bit value within the layer's headers like IP address, checksum, packet length is to be transferred with MSB byte first.

For example: The consider the little-endian node wants to talk with big endian node.

Little endian node:
Would convert the IP address **192.168.0.1** to the little-endian integer **0x0100A8C0.**
The byte transmission will be in order **01 00 A8 C0**
Big endian node:
Would receive the bytes in order **01 00 A8 C0**
This will reconstruct the bytes to big endian integer as **0x0100A8C0**
The IP address will be misinterpreted as **1.0.168.192**
For a stack to be portable it is must to decide whether to do this reordering during compile time.
To do these conversions sockets provide macros to convert to & from the host to network byte order.
To do these conversions sockets provide macros to convert to & from the host to network byte order.

Macro	Description
htons()	**Host to network short** - Reorder bytes of 16-bit unsigned value from the processor to network order.
htonl()	**Host to network long** - Reorder bytes of 32-bit unsigned value from the processor to network order.
ntohs()	**Network to host short** - Reorder bytes of 16-bit unsigned value from the network to processor order.
ntohl()	**Network to host long** - Reorder bytes of 32-bit unsigned value from the network to processor order.

```c
#include <stdio.h>
#include <arpa/inet.h>
int main() {
    int i;
    int Value = 0x0100A8C0; // Value
    unsigned char *ptr1 = (char *) &Value; // Byte pointer

    /* Value in host byte order */
    printf("Value in hex: %02X\r\n", Value);
    printf("Value by bytes: ");

    for (i=0; i < sizeof(long); i++)
        printf("%02X\t", ptr1[i]);
    printf("\r\n");

    /* Value in network byte order */
    Value = htonl(Value);
    printf("\r\nAfter htonl()\r\n");
    printf("Value in hex: %02X\r\n", Value);
    printf("Value by bytes: ");

    for (i=0; i < sizeof(long); i++)
        printf("%02X\t", ptr1[i]);
    printf("\r\n");

    return 0;
}
```

Consider the Example:

test_network_byte_order.c

It shows how the **int Value** with value **0x0100A8C0** (hex) is stored.

Output observations:

When the code is executed on **little endian CPU** check the byte order, we get **LSB 0xC0** in the lowest address **ptr1[0]**.

After use of **htonl()** to convert the **host to network order** we get the **MSB 0x01** in the lowest address **ptr1[0].**

But if you print the **Value** after conversion, you get a meaningless number.

```
akshay@akshayv:Chapter4_Advanced_Data_Types$ gcc test_network_byte_order.c -o test_network_byte_order
akshay@akshayv:Chapter4_Advanced_Data_Types$ ./test_network_byte_order
Value in hex: 0x100A8C0
Value by bytes: C0      A8      00      01      04      00      00      00

After htonl()
Value in hex: 0xC0A80001
Value by bytes: 01      00      A8      C0      04      00      00      00
akshay@akshayv:Chapter4_Advanced_Data_Types$
```

4.1.8 Variable Length Array (VLA)

Is an array data structure whose length is determined at run time (instead of at compile time). In C, the VLA is said to have a variably modified type that depends on a value. Variable length arrays is a feature where we can allocate an auto array (on stack) of variable size. C supports variable sized arrays from C99 standard.

Example:

One Dimensional Array

int n =10;

// Array size will depend on value of **n,** in this case **size = 10**

int arr[n];

Two-Dimensional Array

int row = 3, col = 3;

int arr[row][col]; // Size = 9

```
akshay@akshayv:Chapter4_Advanced_Data_Types$ gcc test_variable_length_arr.c -o test_variable_length_arr
akshay@akshayv:Chapter4_Advanced_Data_Types$ ./test_variable_length_arr
Enter size of one-dimensional array:
5
Enter number of rows & columns of 2-D array:
4
5
One-dimensional array:
a[0] : 0
a[1] : 2
a[2] : 4
a[3] : 6
a[4] : 8
Two-dimensional array:

   0   1   2   3   4
   1   2   3   4   5
   2   3   4   5   6
   3   4   5   6   7
akshay@akshayv:Chapter4_Advanced_Data_Types$
```

Example Program: Demo for usage of Variable length array.

test_variable_length_arr.c

4.1.9 Flexible Array Members (FAM)

Flexible Array Member (FAM) is a feature introduced in the C99 standard of the C programming language. For the structures in C programming language from C99 standard onwards, we can declare an array without a dimension and whose size is flexible in nature. Such an array inside the structure should preferably be declared as the **last member** of structure and its size is variable The structure must contain at least one more named member in addition to the flexible array member. It is declared with an empty index: **datatype arrayName[ ];**

Example:

```
    struct User
{
  int Id;
  int name []; // Flexible array members
 };
```

The size of structure is = 4 + 0 = 4

```
akshay@akshayv:Chapter4_Advanced_Data_Types$ gcc test_flexible_array_member.c -o test_flexible_array_member
akshay@akshayv:Chapter4_Advanced_Data_Types$ ./test_flexible_array_member
Book_cost : 1000
Book_Name : Small World by Laura Zigman (Jan. 10)

Name_Length: 37
Allocated_Struct_size: 49

Book_cost : 2000
Book_Name : Maame by Jessica George (Feb. 7)

Name_Length: 32
Allocated_Struct_size: 44

Size of Struct Book: 12
Size of Struct pointer: 8
akshay@akshayv:Chapter4_Advanced_Data_Types$
```

Example Program: Usage and memory allocation of Flexible Array members.

test_flexible_array_member.c

5. Referencing Data and Functions

Key Learning Outcomes

At the end of this module, you will be able to:

1. Apply the pointers in the code.
2. Apply the binding functions in C code.

UNIT 5.1: Various Pointers

Unit Objectives

At the end of this unit, you will be able to:

1. Apply pointers to data structures, function pointers, pointer and void pointers in C code.

5.1.1 Flexible Array Members (FAM)

A **pointer** is a variable that stores the memory address of another variable as its value. A **pointer variable points to a data type** (like int) of the same type, created with the * operator. We can get the **memory address** of a variable with the reference operator. Using pointers significantly improves performance for repetitive operations, like traversing iterable data structures. Eg: strings, lookup tables, control tables and tree structures.

Syntax:

<dataType> * <pointerName>;

Example:

int myVariable = 85; // Variable declaration

int *ptr = &myVariable; // Pointer declaration

We can use pointers will all the possible datatypes: char, int, array, structure, float, tree, lists etc.

Example Program: How to use pointer.

test_pointers.c

Variable	Value	Address
myVariable	85	0x7ffe65119a0c
ptr	0x7ffe65119a0c	0x7ffe65119a10
***ptr**	85	--

```
akshay@akshayv:Chapter5_Referencing_data_and_functions$ gcc test_pointers.c -o test_pointers
akshay@akshayv:Chapter5_Referencing_data_and_functions$ ./test_pointers
Address of ptr = 0x7ffe65119a10
Address of myVariable = 0x7ffe65119a0c
Value at ptr = 0x7ffe65119a0c
Value at myVariable = 85
Value at *ptr = 85
akshay@akshayv:Chapter5_Referencing_data_and_functions$
```

5.1.2 Pointers to Data Structures

A structure pointer is defined as the pointer which points to the address of the memory block that stores a structure. Complex data structures like **Linked lists**, **trees**, **graphs**, etc. are created with the help of structure pointers. The structure pointer tells the address of a structure in memory by pointing the variable to the structure variable.

The members of struct can be accessed in two ways:

- With the help of (*) asterisk or indirection operator and (.) dot operator.
- With the help of (->) Arrow operator.

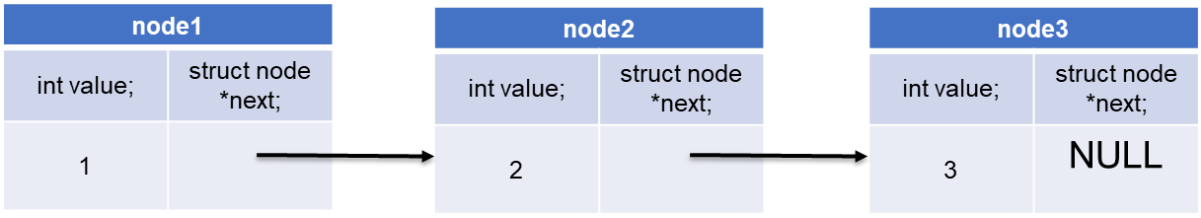
```
struct marks {
   int math;
   int history;
}

struct marks Roll1 = { 83, 75 };
/* p is a pointer to Roll1 */
struct marks *p = &Roll1;
/* set the first member of the struct */
(*p).math = 89;
/* equivalent method to set the first member of the struct */
p->math = 89;
```

Example Program: Usage of Structure Ptr test_structure_ptr.c

```
akshay@akshayv:Chapter5_Referencing_data_and_functions$ gcc test_structure_ptr.c -o test_structure_ptr
akshay@akshayv:Chapter5_Referencing_data_and_functions$ ./test_structure_ptr
Display The Data:
Roll Number: 34
Name: Akshay Varpe
Branch: Electronics & Telecommunication
Batch: 2017
akshay@akshayv:Chapter5_Referencing_data_and_functions$
```

Linked List:

- One practical use of struct pointers is Linked List.
- Consider struct node shown in example.
- Member **struct node *next** is struct ptr.
- So, we use this member to save address of next node in list.
- Let's say we have 3 nodes in the list.



```c
// Creating a node
struct node {
  int value;
  struct node *next;
};
```

Example Program: Linked List implementation. **test_struct_ptr_linked_list.c**

```
akshay@akshayv:Chapter5_Referencing_data_and_functions$ gcc test_struct_ptr_linked_list.c -o test_struct_ptr_linked_list
akshay@akshayv:Chapter5_Referencing_data_and_functions$ ./test_struct_ptr_linked_list
1
2
3
akshay@akshayv:Chapter5_Referencing_data_and_functions$
```

5.1.3 Function Pointers

As opposed to referencing a data value, a function pointer points to executable code within memory. Dereferencing the function pointer yields the referenced function, which can be invoked and passed arguments just as in a normal function call. Such an invocation is also known as an "indirect" call, because the function is being invoked indirectly through a variable instead of directly through a fixed identifier or address. Function pointers can be used to simplify code by providing a simple way to select a function to execute based on run-time values.

Syntax:

```c
// Function to add two numbers:
int add(int a, int b)
{
    return a + b;
}

// Function pointer:
< func return type >      < ptr >     < func args >     < function to point >
    int                   (*fun_ptr)  (int, int)      = add;
```

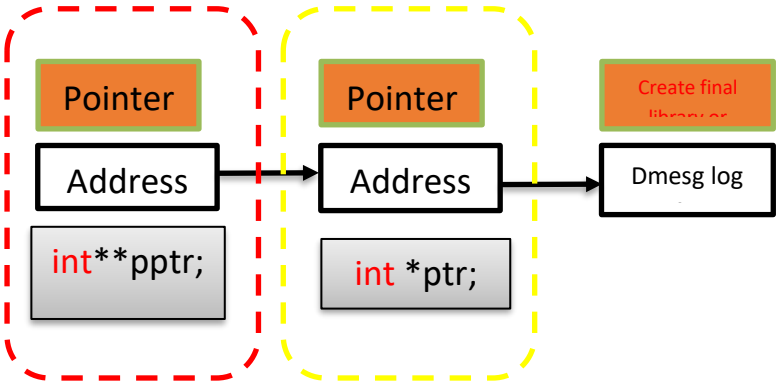
```
akshay@akshayv:Chapter5_Referencing_data_and_functions$ gcc test_function_ptr.c -o test_function_ptr
akshay@akshayv:Chapter5_Referencing_data_and_functions$ ./test_function_ptr
Variable values:
 a = 15
 b = 10
Enter Choice:
 0 -> Addition
 1 -> Subtract
 2 -> Multiply
0
Choice = 0
Addition is 25
akshay@akshayv:Chapter5_Referencing_data_and_functions$ ./test_function_ptr
Variable values:
 a = 15
 b = 10
Enter Choice:
 0 -> Addition
 1 -> Subtract
 2 -> Multiply
1
Choice = 1
Subtraction is 5
akshay@akshayv:Chapter5_Referencing_data_and_functions$ ./test_function_ptr
Variable values:
 a = 15
 b = 10
Enter Choice:
 0 -> Addition
 1 -> Subtract
 2 -> Multiply
2
Choice = 2
Multiplication is 150
akshay@akshayv:Chapter5_Referencing_data_and_functions$
```

Example Program: Use of function ptr to execute function based on choice.

test_function_ptr.c

5.1.4 Pointer to Pointer

A pointer to a pointer is a form of multiple indirections, or a chain of pointers. When we define a pointer to a pointer, the first pointer contains the address of the second pointer, which points to the location that contains the actual value as shown in the image. A variable that is a pointer to a pointer must be declared is done by placing an additional asterisk in front of its name. When a target value is indirectly pointed to by a pointer to a pointer, accessing that value requires that the asterisk operator be applied twice.



```
akshay@akshayv:Chapter5_Referencing_data_and_functions$ gcc test_ptr_to_ptr.c -o test_ptr_to_ptr
akshay@akshayv:Chapter5_Referencing_data_and_functions$ ./test_ptr_to_ptr
Value of var = 4000
Value available at *ptr = 4000
Value available at **pptr = 4000
akshay@akshayv:Chapter5_Referencing_data_and_functions$
```

Example Program: Use of Pointer to Pointer.**test_ptr_to_ptr.c**

5.1.5 Void Printers

A void pointer is a pointer that has no associated data type with it. A void pointer can hold address of any type and can be typecasted to any type. In C, malloc () and calloc() functions return void * or generic pointers.

Syntax:

void *name_of_pointer;

One can assign the void pointer with any data type's address, and then assign the void pointer to any pointer without even performing some sort of explicit typecasting. So, it reduces complications in a code.

```
#include <stdlib.h>
#include <stdio.h>
int main()
{
        // Integer variable
        int v = 7;
        // Float variable
        float w = 7.6;
        // void Pointer
        void *u;
        u = &v; // Pointer to Integer variable
        printf("The Integer variable = % d\r\n", *((int
        *)u));
        u = &w; // Pointer to Float variable
        printf("The Float variable = % f\r\n", *((float *)u));
        return 0;
        }
```

Output:

```
The Integer variable = 7

The Float variable = 7.600000
```

5.1.6 Binding of Variables and Functions

A binding is the attachment of a name (also called an identifier) to a variable type and function in your program. Binding refers to the process of converting identifiers into addresses. Binding is done for each variable and function. For functions, it means that matching the call with the right function definition by the compiler. Binding is intimately connected with scoping, as scope determines which names bind to which objects – at which locations in the program code (statically) and in which one of the possible execution paths temporally. It takes place either at compile time or at runtime. Binding finds the corresponding binding occurrence (declaration/definition) for an applied usage of an identifier. For binding there are some points that need to be considered: The scope of variables should be known. What will happen if we use same identifier name again? 'C' forbids use of the same identifier name in the same scope. Same name can be used in different scope.

Scope:

Scope in any programming language is a region of program where a defined variable/function can have its existence. Beyond that block/file the variable/function cannot be accessed. A Block is a set of statements enclosed within left and right braces i.e. '{' and '}' respectively. This variable/function has a File Scope, i.e., we can access the anywhere in the same C file but we cannot access outside that C file.

Scope	Place
Local Variable	Inside a function or block.
Global Variable	Outside of all function (can be accessed from anywhere) in the program
Formal Parameters	In the function parameters. Formal parameters are the parameter that are written in the function definition.
Static	Inside a block or function.
Static Global	In all the functions in the C file. But not outside that C file

Binding Time:

At what time a value is assigned an attribute. There are two types of binding: Static binding and dynamic binding.

Static/Early Binding:

A binding is static if it first occurs before run time and remains unchanged throughout program execution. Static binding occurs at compile time. All the final, static, and private methods are bound at run time. Program execution is faster. In C, there is only static binding.

Example Program: **Static Binding**

Directory **"Module3_SystemProgramming_Using_C/Chapter5_Referencing_data_and_functions/StaticBindingDemo "** contains source code to demonstrate static binding with different scope of variables and functions.

Compile the code: **gcc main.c file1.c -o main**

We have one global variable in main.c :

int global = 5;

fille1.c & file1.h have **ModifyTheVariable** function to modify a value and is declared into file1.h file

In main.c & file1.c we have static variable & function with same name. As they are in different file scope it is allowed to have same names to the identifiers.

StaticVariable

AlterTheValueWithStatic

In main (), we have one code **Block A:**

- Inside Block A we have declared one more int variable: **int global = 100;**
- From output the variable changes are valid only inside the Block A

```
akshay@akshayv:BindingDemo$ ls
file1.c  file1.h  main.c
akshay@akshayv:BindingDemo$ gcc main.c file1.c -o main
akshay@akshayv:BindingDemo$ ls
file1.c  file1.h  main  main.c
akshay@akshayv:BindingDemo$ ./main
[main.c:31][main]Before change within main:
[main.c:32][main]Value of global = 5
[main.c:24][display]Value of global = 5
[main.c:37][main]After change within main:
[main.c:24][display]Value of global = 10
[main.c:43][main]-----Inside Block A------:
[main.c:44][main]Value of global = 100
[main.c:24][display]Value of global = 10
[file1.c:16][ModifyTheVariable]Value of a = 100
[file1.c:9][AlterTheValueWithStatic]Value of a = 100
[file1.c:10][AlterTheValueWithStatic]Value of -> a * StaticVariable = 1000
[main.c:48][main]-----Block A Ends-----
[main.c:50][main]After change within Block A:
[main.c:24][display]Value of global = 10
[main.c:15][AlterTheValueWithStatic]Value of a = 10
[main.c:16][AlterTheValueWithStatic]Value of -> a * StaticVariable = 500
[main.c:56][main]After change value change using static function:
[main.c:24][display]Value of global = 500
akshay@akshayv:BindingDemo$
```

The output shown above is formatted as follows: [<filename>:<Line Number>][Function Name]<Print Message>

Dynamic/Late binding:

A binding is dynamic if it first occurs during execution or can change during execution of the program. Dynamic binding occurs at Run time. The compiler doesn't decide the method to be called. Based on the type of object, the respective function will be called. Dynamic binding helps us to handle different objects using a single function name. It also reduces complexity and helps the developer to debug the code and errors. In simple words, dynamic binding is just delaying the choice or selection of which function to run until its runtime. Program execution is slower. The concept of dynamic programming is implemented with virtual functions.

Virtual Functions:

Virtual functions are special member functions to which calls are made through a pointer (or reference) are resolved at run time, based on the object's type with the pointer. A function declared in the base class and overridden (redefined) in the child class is called a virtual function.

When we refer derived class object using a pointer or reference to the base, we can call a virtual function for that object and execute the derived class's version of the function.

Example Program:

Directory **"Module3_SystemProgramming_Using_C/Chapter5_Referencing_data_and_functions/DynamcBindingDemo "** contains source code to demonstrate Static binding with different scope of variables and functions.

Consider the program: **test_Without_DynamicBinding.cpp**

Class Base_A with a function print_info(), and class Derived_B inherits Base_A publicly.

Derived_B also has its print_info() function.

If we make an object of Base_A and call print_info(), it will run of base class whereas, if we make an object of Derived_B and call print_info(), it will run of base only.

```
akshay@akshayv:DynamicBindingDemo$ ls
test_DynamicBinding.cpp  test_Without_DynamicBinding.cpp
akshay@akshayv:DynamicBindingDemo$ g++ test_Without_DynamicBinding.cpp -o test_Without_DynamicBinding
akshay@akshayv:DynamicBindingDemo$ ls
test_DynamicBinding.cpp  test_Without_DynamicBinding  test_Without_DynamicBinding.cpp
akshay@akshayv:DynamicBindingDemo$ ./test_Without_DynamicBinding
Printing from the Base_A class
Printing from the Base_A class
akshay@akshayv:DynamicBindingDemo$
```

```cpp
#include <iostream>
using namespace std;
class Base_A
{
public:
    void print_info() // function that call display
    {
        display();
    }
    void display() // the display function
    {
        cout << "Printing from the Base_A class" << endl;
    }
};
class Derived_B : public Base_A // Derived_B inherit a publicly
{
public:
    void display() // Derived_B's display function
    {
        cout << "Printing from the Derived_B class" << endl;
    }
};
int main()
{
    Base_A obj1;        // Creating Base_A's pbject
    obj1.print_info(); // Calling final_print
    Derived_B obj2;     // Creating Derived_B
    obj2.print_info();
    return 0;
}
```

Example Program:

Consider the program: test_DynamicBinding.cpp

Class Base_A with a function print_info(), and class Derived_B inherits Base_A publicly.

Derived_B also has its print_info() function.

Base_A and Derived_B also has its display () function with virtual keyword. If we make an object of Base_A and call print_info(), it will run from the base class, if we make an object of Derived_B and call print_info(), it will run display() of derived class. This way dynamic binding links the function call

with function definition with the help of virtual functions.

```
akshay@akshayv:DynamicBindingDemo$ ls
test_DynamicBinding.cpp  test_Without_DynamicBinding  test_Without_DynamicBinding.cpp
akshay@akshayv:DynamicBindingDemo$ g++ test_DynamicBinding.cpp -o test_DynamicBinding
akshay@akshayv:DynamicBindingDemo$ ls
test_DynamicBinding  test_DynamicBinding.cpp  test_Without_DynamicBinding  test_Without_DynamicBinding.cpp
akshay@akshayv:DynamicBindingDemo$ ./test_DynamicBinding
Printing from the Base_A class
Printing from the Derived_B class
akshay@akshayv:DynamicBindingDemo$
```

```cpp
#include <iostream>
using namespace std;
class Base_A
{
public:
    void print_info() // function that call display
    {
        display();
    }
    virtual void display() // the display function
    {
        cout << "Printing from the Base_A class" << endl;
    }
};
class Derived_B : public Base_A // Derived_B inherit a publicly
{
public:
    virtual void display() // Derived_B's display function
    {
        cout << "Printing from the Derived_B class" << endl;
    }
};
int main()
{
    Base_A obj1;        // Creating Base_A's pbject
    obj1.print_info(); // Calling final_print
    Derived_B obj2;     // Creating Derived_B
    obj2.print_info();
    return 0;
}
```

6. Working with Registers

Unit 6.1 - Bit Level Operation

Unit 6.2 – Handling Special Registers

Key Learning Outcomes

At the end of this module, you will be able to:

1. Implement the bit level operations in the code.
2. Develop a code by handling the registers.

UNIT 6.1: Bit Level Operations

Unit Objectives

At the end of this unit, you will be able to:

1. Apply the bit mask operation in the code
2. Apply the bitwise shift operation in the code
3. Apply the structure and bit fields operation in the code

6.1.1 Bit Level Operations

In C programming, operations can be performed on a bit level using bitwise operators. Bitwise operations are contrasted by byte-level operations. Instead of operations on individual bits, byte-level operators perform on strings of eight bits (known as bytes) at a time. The reason for this is that a byte is normally the smallest unit of addressable memory. C provides six operators for bit manipulation:

Operator	Description	Example
&	Binary AND Operator copies a bit to the result if it exists in both operands.	(A & B) = 12, i.e., 0000 1100
|	Binary OR Operator copies a bit if it exists in either operand.	(A | B) = 61, i.e., 0011 1101
^	Binary XOR Operator copies the bit if it is set in one operand but not both.	(A ^ B) = 49, i.e., 0011 0001
~	Binary One's Complement Operator is unary and has the effect of 'flipping' bits.	(~A) = ~(60), i.e.,. 1100 0011
<<	Binary Left Shift Operator. The left operands value is moved left by the number of bits specified by the right operand.	A << 2 = 240 i.e., 1111 0000
>>	Binary Right Shift Operator. The left operands value is moved right by the number of bits specified by the right operand.	A >> 2 = 15 i.e., 0000 1111

Consider the Program: **test_bitwise_operations.c**. The program shows the use of bit level operators

```
akshay@akshayv:Chapter6_Working_with_registers$ gcc test_bitwise_operations.c  -o test_bitwise_operations
akshay@akshayv:Chapter6_Working_with_registers$ ./test_bitwise_operations
a = 60
b = 13

AND:     a & b = 12
OR:      a | b = 61
XOR:     a ^ b = 49
One's Compliment: ~a = -61
Left Shift: a << 2 = 240
Right Shift: a >> 2 = 15
akshay@akshayv:Chapter6_Working_with_registers$
```

```c
#include <stdio.h>

int main()
{

    unsigned int a = 60; /* 60 = 0011 1100 */
    unsigned int b = 13; /* 13 = 0000 1101 */
    int c = 0;

    printf("a = %d\r\n", a);
    printf("b = %d\r\n\r\n", b);
    c = a & b; /* 12 = 0000 1100 */
    printf("AND:\t a & b = %d\r\n", c);

    c = a | b; /* 61 = 0011 1101 */
    printf("OR:\t a | b = %d\r\n", c);

    c = a ^ b; /* 49 = 0011 0001 */
    printf("XOR:\t a ^ b = %d\r\n", c);

    c = ~a; /*-61 = 1100 0011 */
    printf("One's Compliment: ~a = %d\r\n", c);

    c = a << 2; /* 240 = 1111 0000 */
    printf("Left Shift: a << 2 = %d\r\n", c);

    c = a >> 2; /* 15 = 0000 1111 */
    printf("Right Shift: a >> 2 = %d\r\n", c);

    return 0;
}
```

6.1.2 Bitwise Shift Operations

The bitwise shift operators move the bit values of a binary object. The left operand specifies the value to be shifted. The right operand specifies the number of positions that the bits in the value are to be shifted. The result is not a l value. Both operands have the same precedence and are left-to-right associative. Operator **<<** Indicates the bits are to be **shifted to the left**. Operator **>>** Indicates the bits are to be **shifted to the right**. The right operand should not have a negative value or a value that is greater than or equal to the width in bits of the expression being shifted. The result of bitwise shifts on such values is unpredictable. If the right operand has the value 0, the result is the value of the left operand. The << operator fills vacated bits with zeros.

Consider the example program: **test_ShiftOperators.c**. It shows how to use the Shift operators.

```
akshay@akshayv:Chapter6_Working_with_registers$ gcc test_ShiftOperators.c -o test_ShiftOperators
akshay@akshayv:Chapter6_Working_with_registers$ ./test_ShiftOperators
a =             0b00000000000000000000000000000101
a << 1 =        0b00000000000000000000000000001010

b =             0b00000000000000000000000000001001
b << 4 =        0b00000000000000000000000010010000

a =             0b00000000000000000000000000000101
a >> 1 =        0b00000000000000000000000000000010

b =             0b00000000000000000000000000001001
b >> 2 =        0b00000000000000000000000000000010

akshay@akshayv:Chapter6_Working_with_registers$
```

```
// Shift Left Lnum number of bit positions:
#define SHIFT_LEFT(x, Lnum) (x << Lnum)


// Shift Right Rnum number of bit positions:
#define SHIFT_RIGHT(x, Rnum) (x >> Rnum)
```

6.1.3 Bitmasks

In computer science, a bitmask is data that is used for bitwise operations, particularly in a bit field. Using a mask, multiple bits in a byte, nibble, word, etc. can be set either on or off, or inverted from on to off (or vice versa) in a single bitwise operation. The idea for bit masking is based on Boolean logic. One of these true/false values is a bit.

Primitives in C (int, float, etc.) are made up of some number of bits:

Char - 1 Byte - 8 bits

Int - 4 Bytes - 32 bits

Uint16_t - 2 Bytes - 16 bits

Masking is a general concept in which we keep, change, or remove some part of the information.

Common Bitmask functions:

Masking bits to 1:

To turn certain bits on, the bitwise OR operation can be used. When a bit A OR 1 = 1 and A OR 0 = A. Therefore, to make sure a bit is on, OR can be used with a 1. To leave a bit unchanged, OR is used with a 0.

Example: Masking bits 4,5,6,7 to 1 while leaving bits 0, 1,2,3 unchanged.

	10010101	**10101010**
OR	11110000	11110000
=	11110101	11111010

Masking bits to 0:

To "masked off" (or masked to 0) than "masked on" (or masked to 1). When a bit is ANDed with a 0, the result is always 0, i.e. A AND 0 = 0. To leave the other bits as they were originally, they can be ANDed with 1 as A AND 1 = A

Example: Masking off bits 4,5,6,7 and while leaving bits 0, 1,2,3 unchanged.

	10010101	**10101010**
AND	00001111	00001111
=	00000101	00001010

Querying the status of a bit:

It is possible to use bitmasks to easily check the state of individual bits regardless of the other bits. To do this turning off all the other bits using the bitwise AND is done. The value is compared with 0. If it is equal to 0, then the bit was off, but if the value is any other value, then the bit was on.

	10010101	**10010101**
AND	00010000	00001000
=	00010000	00000000

Toggling bit values:

To make the bits opposite of what it currently is it called toggling. This can be achieved using the XOR (exclusive or) operation. XOR returns 1 if and only if an odd number of bits are 1. Therefore, if two corresponding bits are 1, the result will be a 0, but if only one of them is 1, the result will be 1. If the original bit was 1, it returns **1 XOR 1 = 0**. If the original bit was 0 it returns **0 XOR 1 = 1**.

	10011101	10010101
XOR	**00001111**	**11111111**
=	10010010	01101010

Consider the example program: **test_bitmasks.c**

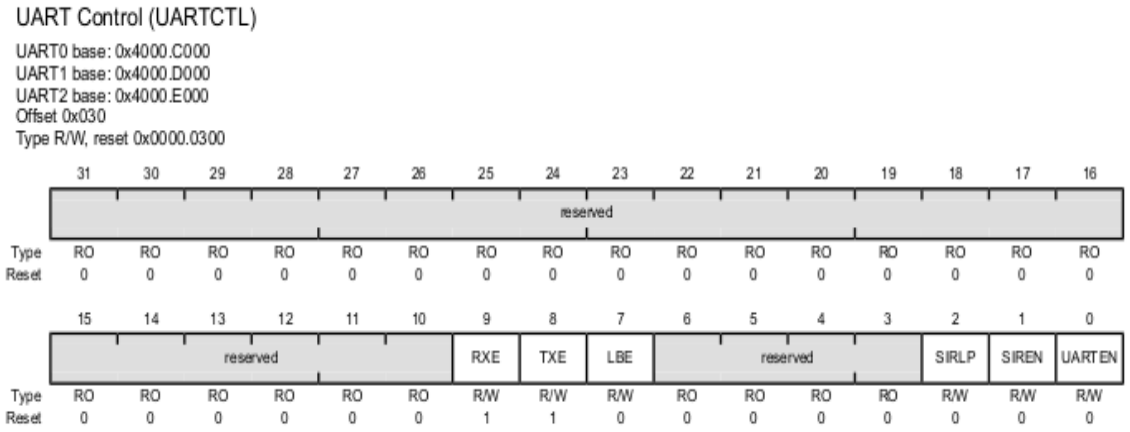
The program shows how to use different bitwise operation to operate in bits.

6.1.4 Accessing Registers

Device Registers:

A peripheral device is likely to have a number of internal registers, which may be read from or written by software. These normally appear just like memory locations and can, for the most part, be treated in the same way. Typically, a device register will have bit fields - groups of bits that contain or receive specific information. Such fields may be single bits, groups of bits, or a whole word. There may also be bits that are unused - reading from them or writing to them normally has no effect.

For example: Consider UART Device Control Register



Addressing the Register

Typically, when we access registers in C based on memory-mapped IO we use a pointer notation. Its 'trick' the compiler into generating the correct load/store operations at the absolute address needed.

For Example: UART0 UARTCTL register:

The peripheral Base Address for UART0 = 0x4000C000

The Register Offset = 0x030

This means UART0 UARTCTL register is located at 0x4000C030

The declaration of the pointer could include an initialization, as:

volatile uint32_t* const Uart0_UARTCTL_reg = ((uint32_t*) 0x4000C030);

#Define UART0_UARTCTL_REG (*((volatile uint32_t*) 0x4000C030))

Reading the Register

Using the pointer definition from last slide we can read the reg data.

uint32_t UartCTL_Val = UART0_UARTCTL_REG;

Writing to Registers:

Consider we want to want to set following bits in the UARTCTL reg

RXE - bit 9

TXE - bit 8

UARTEN - bit 0

All the Bitwise operators can be used to manipulate the bits.

```
// Read the value

uint32_t UartCTL = UART0_UARTCTL_REG;

/* Set the bits 9, 8 & 0*/

UartCTL |= (1 << 9); // Set RXE

UartCTL |= (1 << 8); // Set TXE

UartCTL |= (1 << 0); // Set UARTEN

// Write value to register:

UART0_UARTCTL_REG = UartCTL;
```

6.1.5 Structure Padding

Structure members are assigned to memory addresses in increasing order, with the first component starting at the beginning address of the structure name itself. Structure padding is a concept in C that adds one or more empty bytes between the memory addresses to align the data in memory. The processor does not read 1 byte at a time. It reads 1 word at a time.

What does the 1 word mean? If we have a **32-bit processor**, then the processor reads 4 bytes at a time **1 word = 4 bytes.** If we have a **64-bit processor**, then the processor reads 8 bytes at a time, **1 word = 8 bytes.** As we know that structure occupies the contiguous block of memory as shown in diagram.

Consider the **32bit architecture:**

The problem is that in one CPU cycle we can access:

- one byte of **char a**,
- one byte of **char b**,
- bytes of **int c**.

We will not face any problem while accessing the char a and char b as both the variables can be accessed in one CPU Cycle. We will face the problem when we access the int c variable as 2 CPU cycles are required to access the value of the 'c' variable.

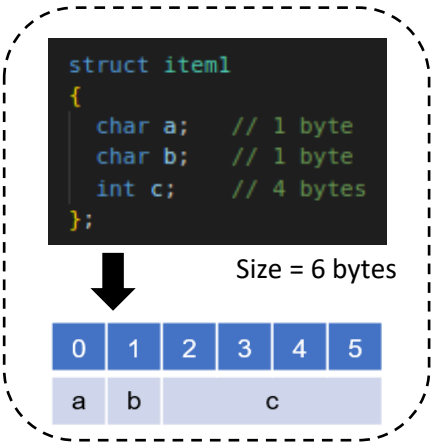


Fig 6.1.5 Structure Padding

Suppose we only want to access the variable 'c', which requires two cycles. The variable 'c' is of 4 bytes, so it can be accessed in one cycle also, but in this scenario, it is utilizing 2 cycles. This is an unnecessary wastage of CPU cycles. Due to this reason, the structure padding concept was introduced to save the number of CPU cycles.

How is structure padding done?

To achieve the structure padding, an empty row is created on the left, as shown in the diagram. The two bytes which are occupied by the 'c' variable on the left are shifted to the right. So, all four bytes of 'c' variable are on the right. Now, the 'c' variable can be accessed in a single CPU cycle. After structure padding, the total memory occupied by the structure is 8 bytes (1 byte +1 byte +2 bytes +4 bytes). Although the memory is wasted in this case, the variable can be accessed within a single cycle.

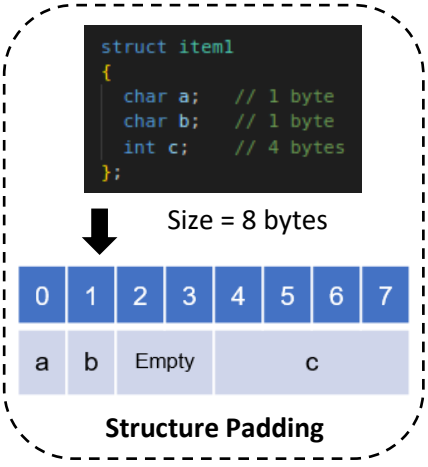


Fig 6.1.5 Structure Padding

The structural padding is an in-built process that is automatically done by the compiler. Sometimes it required to avoid the structure padding in C as it makes the size of the structure greater than the size of the structure members.

Packing:

Packing, prevents the compiler from doing padding means remove the unallocated space allocated by structure. We can avoid the structure padding in C in two ways:

Using #pragma pack (1) directive

Using __attribute__((__packed__))

```c
// To avoid the structure padding.
// Use the #pragma pack(1) directive
#pragma pack(1)
struct item3
{
    char a;  // 1 byte
    char b;  // 1 byte
    int  c;  // 4 byte
};

// By using attribute
struct item4
{
    char a;  // 1 byte
    char b;  // 1 byte
    int  c;  // 4 bytes
}__attribute__((packed));;
```

Example Program: Demo for struct padding:

test_structure_padding.c

```
akshay@akshayv:Chapter4_Advanced_Data_Types$ gcc test_structure_padding.c -o test_structure_padding
akshay@akshayv:Chapter4_Advanced_Data_Types$ ./test_structure_padding
The size of the item1 structure = 8
The size of the item2 structure = 12
The size of the item3 structure = 6
The size of the item4 structure = 6
akshay@akshayv:Chapter4_Advanced_Data_Types$
```

6.1.6 Bit Fields, Structures, and Union

Bit Fields:

A bit field is a data structure that consists of one or more adjacent bits which have been allocated for specific purposes, so that any single bit or group of bits within the structure can be set or inspected. A bit field is most commonly used to represent integral types of known, fixed bit-width, such as single-bit Booleans. Bit fields can be used to reduce memory consumption when a program requires a number of integer variables, which will always have low values. For example, in many systems storing an integer value require two bytes (16-bits) of memory; sometimes the values to be stored actually need only one or two bits. Having a number of these tiny variables share a bit field allows efficient packaging of data in the memory.

In C, we can specify the size (in bits) of the structure and union members.

Declaration of bit-fields in C

Syntax:

```
struct
{
   data_type member_name : width_of_bit-field;
};
```

Consider the sample program: **test_bitfeilds.c**

It shows use of bit fields for efficient packaging of data in the memory.

```
akshay@akshayv:Chapter6_Working_with_registers$ gcc test_bitfeilds.c  -o test_bitfeilds
akshay@akshayv:Chapter6_Working_with_registers$ ./test_bitfeilds
 Simple Date representation
 Size of Simpledate is 12 bytes
 Date is 31/12/2014

 Bitfeild Date representation
 Size of date is 6 bytes
 Date is 31/12/2014
akshay@akshayv:Chapter6_Working_with_registers$
```

```c
// Simple representation of the date
struct Simpledate
{
    unsigned int Date;   // 4 Bytes
    unsigned int Month;  // 4 Bytes
    unsigned int Year;   // 4 Bytes
};

// Space optimized representation of the date
struct date
{
    // Date -> between 0 and 31,
    // so 5 bits are sufficient
    unsigned int Date : 5;
    //  Month has value between 0 and 15,
    // so 4 bits are sufficient
    unsigned int Month : 4;
    // Year
    unsigned int Year; // 4 Bytes
};
```

Bitfeilds can be used for accessing the device registers in C.

For Example: **Bitfeild_UARTCTL_REG.c**

The code shows the representation of UARTCTL 32bit register (4 bytes).

```
// Structure defining the register UARTCTL:
typedef struct
{
    unsigned int UARTEN     : 1;    // UARTEN   Bit  0
    unsigned int SIREN      : 1;    // SIREN    Bit 1
    unsigned int SIRLP      : 1;    // SIRLP    Bit 2
    unsigned int reserved1  : 4;    // Reserved Bits 3: 6
    unsigned int LBE        : 1;    // LBE      Bit 7
    unsigned int TXE        : 1;    // TXE      Bit 8
    unsigned int RXE        : 1;    // RXE      Bit 9
    unsigned int reserved2  : 22;   // Reserved Bits 10 : 31
} UARTCTL_BITS;

// Union of Int and Structure Reg Bits
typedef union
{
    unsigned int UARTCTL_Reg;
    UARTCTL_BITS BitsRep;
} UARTCTL_reg_t;
```

We can manipulate bits by accessing the structure fields

```
// Set the bits 9, 8 & 0
RegData.BitsRep.RXE = 1;
RegData.BitsRep.TXE = 1;
RegData.BitsRep.UARTEN = 1;
```

Using the Device register pointer. We can write data to register as below:

```
// Uart0 UARTCTL Reg Addr for device register map
#define UART0_UARTCTL_REG (*((volatile uint32_t*)  0x4000C030))
```

```
// To write the new data to Device register:
// Use the pointer defination we got from previous section:
UART0_UARTCTL_REG = RegData.UARTCTL_Reg;
```

```
akshay@akshayv:Chapter6_Working_with_registers$ gcc Bitfeild_UARTCTL_REG.c -o Bitfeild_UARTCTL_REG
akshay@akshayv:Chapter6_Working_with_registers$ ./Bitfeild_UARTCTL_REG
Bitfeild Device Register : UARTCTL
Size of UARTCTL_reg_t = 4 bytes

Reset Value of RegData.UARTCTL_Reg
RegData.UARTCTL_Reg =   0b00000000000000000000000000000000

After setting bits 9, 8, 0 Value of RegData.UARTCTL_Reg
RegData.UARTCTL_Reg =   0b00000000000000000000001100000001
akshay@akshayv:Chapter6_Working_with_registers$
```

6.1.7 Processor Endianness

Endianness is a term that describes the order in which a sequence of bytes is stored in computer memory.

Types of Endianness:

Little endian:

Last byte of binary representation of the multibyte data-type is stored first.

X86 processor and most of the ARM Cortex-M3 based microcontrollers use the little-endian format.

Big endian:

First byte of binary representation of the multibyte data-type is stored first.

Motorola 6800 / 6801 processors use the big-endian format.

Example:

Integer is stored as 4bytes. will be stored as following:

Little Endian	
Address	**Data**
0x100	0x67
0x101	0x45
0x102	0x23
0x103	0x01

int x = 0x01234567

Big Endian	
Address	**Data**
0x100	0x01
0x101	0x23
0x102	0x45
0x103	0x67

Most of the times compiler takes care of endianness. When we perform bit-wise operation on integer then compiler automatically handles the endianness.

However, endianness becomes an issue in following cases:

Writing raw bytes of data to a file in one processor and you send it to a system that uses different endian processor. Send bytes of data over network as a serialized steam of data from one endian processor system to other endian processor system. In network communication, TCP/IP suites are defined to be big-endian.

Sometimes it matters when you are using type casting, below program is an example.

Consider the example in image:

A char array is typecasted to an integer type.

On little endian machine we get:

DataInt = 1

On Big endian machine we get:

DataInt = 16777216

Example Program: To find Endianness of system

test_endianness.c

```
akshay@akshayv:Chapter4_Advanced_Data_Types$ gcc test_endianness.c -o test_endianness
akshay@akshayv:Chapter4_Advanced_Data_Types$ ./test_endianness
int Data = 0x01234567

    Address      Data
0x7ffd6d53beb4   0x67
0x7ffd6d53beb5   0x45
0x7ffd6d53beb6   0x23
0x7ffd6d53beb7   0x01
```

Example Program:

test_endianness_issue.c

```c
#include <stdio.h>
int main()
{
    unsigned char DataArr[4] = {0x01, 0x00, 0x00, 0x00 };
    // char array is typecasted to an integer type.
    int DataInt = *(int *) DataArr;
    printf("DataInt = %d\r\n", DataInt);
    return 0;
}
```

```
akshay@akshayv:Chapter4_Advanced_Data_Types$ gcc test_endianness_issue.c -o test_endianness_issue
akshay@akshayv:Chapter4_Advanced_Data_Types$ ./test_endianness_issue
DataInt = 1
akshay@akshayv:Chapter4_Advanced_Data_Types$
```

Sometimes you may need to convert the endianness of data.

Using the union, we change the endianness of data:

Example program:

test_change_endianness_union.c

```
akshay@akshayv:Chapter4_Advanced_Data_Types$ gcc test_change_endianness_union.c -o test_change_endianness_union
akshay@akshayv:Chapter4_Advanced_Data_Types$ ./test_change_endianness_union
Orignal Byte Order    = 0x11223344
Converted Byte Order = 0x44332211
akshay@akshayv:Chapter4_Advanced_Data_Types$
```

6.1.8 Bit Endianness

In computing, bit numbering is the convention used to identify the bit positions in a binary number. Bit numbering is a concept similar to endianness, but on a level of bits, not bytes. The least significant bit (LSB) is the bit position in a binary integer representing the binary 1s place of the integer. Similarly, the most significant bit (MSB) represents the highest-order place of the binary integer. The LSB is sometimes referred to as the low-order bit or right-most bit, due to the convention in positional notation of writing less significant digits further to the right. The MSB is similarly referred to as the high-order bit or left-most bit. In both cases, the LSB and MSB correlate directly to the least significant digit and most significant digit of a decimal integer. Bit indexing correlates to the positional notation of the value in base 2. For this reason, bit index is not affected by how the value is stored on the device, such as the value's byte order.

For Example

Binary (Decimal: 149)	**1**	**0**	**0**	**1**	**0**	**1**	**0**	**1**
Bit weight for given bit position n (2^n)	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
Bit position label	MSB							LSB

UNIT 6.2: Handling Special Registers

Unit Objectives

At the end of this unit, you will be able to:

1 Explain the FIFO operation
2 Explain the data alignment in the registers
3 Use the type qualifiers in the code

6.2.1 FIFO

A FIFO buffer is a useful way to store data that arrives at a microcontroller peripheral asynchronously but cannot be read immediately. An example of this is storing bytes that are incoming on a UART. Buffering the bytes can make it easier for the embedded firmware to handle the incoming data in real time. A FIFO buffer is a type of data storage that operates on a first-in, first-out basis. It is a very common construct used in digital systems, and FIFOs can be implemented with software or hardware. The choice between a software and a hardware solution depends on the application and the features desired.

Fig 6.2.1 FIFO

An example of this is storing bytes that are incoming on a UART. Buffering the bytes can make it easier for the embedded firmware to handle the incoming data in real time. Many microcontroller designs have limited buffer space for data arriving on the UART. Using a FIFO in the UART ISR can make it easier to manage incoming data. Using a FIFO as described above can reduce the real-time requirements for an application, as well as give the application developer more flexibility in handling incoming data. Because the FIFO buffer stores the data until it can be processed by the application, the application does not need to handle each byte as it arrives. This can make it easier to develop real-time applications that must process large amounts of data.

6.2.2 Register Windows

In computer architecture, the registers are very fast computer memory which are used to execute programs and operations efficiently. The sole purpose of having register is fast retrieval of data for processing by the CPU. Register windows are a feature which dedicates registers to a subroutine by dynamically aliasing a subset of internal registers to fixed, programmer-visible registers. Several sets of registers are provided for the different parts of the program. Registers are deliberately hidden from

the programmer to force several subroutines to share processor resources. The CPU recognizes the movement from one part of the program to another during a procedure call. If you consider register organization of RISC CPU, it uses an overlapped register window that provides the passing of parameters to called procedure and stores the result to the calling procedure. For each procedure call, the new register window is assigned from the register file used by the new procedure. Each procedure call activates the new register window by incrementing a pointer, and the return statement decrements the pointer which causes the activation of the previous window. Windows of adjacent procedures have overlapping registers that are shared to provide the passing of parameters and storage of results.

There are three classes of registers: –

- Global Registers: Available to all functions
- Window local registers: Variables local to the function
- Window shared registers: Permit data to be shared without actually needing to copy it.

For Example:

Procedure C called procedure D. Therefore, registers R58 to R63 are common to both procedures C and D. Therefore, procedure C stores parameters for procedure D in these registers. Procedure D uses registers R64 to R73 to store the local variables. When procedure B returns after performing its computation, then the result from registers (R26 to R31) is transferred back to window A. Registers R10 to R15 are common to procedures A and D also the four windows have a circular organization. As R0 to R9 (i.e., 10 registers) is available for all procedures. Therefore, a procedure contains 32 registers while procedure it is active (which includes 10 global, 10 local, and 6 low overlapping registers and 6 highly overlapping registers).

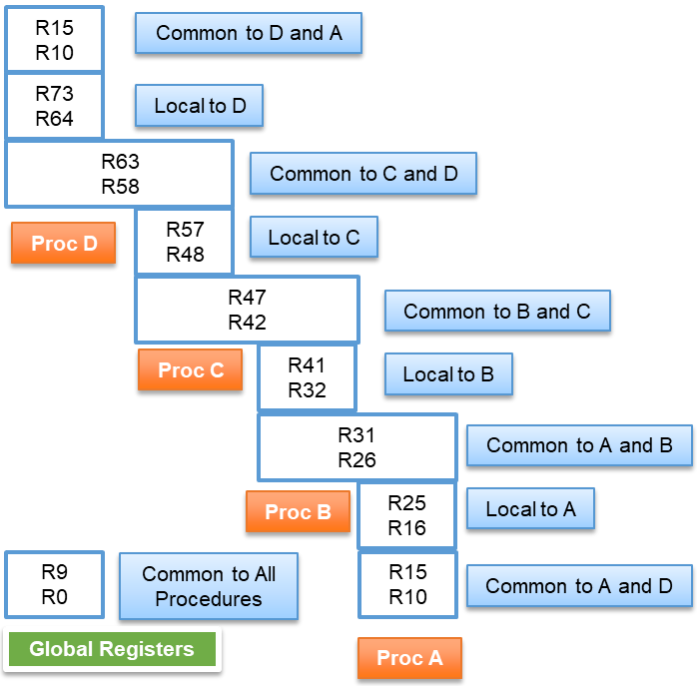


Fig 6.2.2 Registers

6.2.3 Write Only Register

A write-only register is a register that can be written to but not read. In addition to its literal meaning, the term may be applied to a situation when the data written by one circuit can be read only by another circuitry. The most common occurrence of the latter situation is when a processor writes data to a write-only register of hardware the processor is controlling. The hardware can read the instruction, but the processor cannot. For instance, a GPU might be carrying out shader (a computer program that calculates the appropriate levels of light, darkness,) processing on the contents of graphics memory. It can be faster and more efficient for the GPU to take input for the shader process from read-only locations and write the shader output to different write-only locations without having to copy data between the read and write buffers after each iteration.

6.2.4 Alignment

One of the low-level features of C is the ability to specify the precise alignment of objects in memory to take maximum advantage of the hardware architecture. CPUs read and write memory more efficiently when they store data at an address that's a multiple of the data size. For example, a 4-byte integer is accessed more efficiently if it's stored at an address that's a multiple of 4.

When data isn't aligned, the CPU does more address calculation work to access the data. By default, the compiler aligns data based on its size:

- char on a 1-byte boundary
- short on a 2-byte boundary
- int, long, and float on a 4-byte boundary

double on 8-byte boundary, and so on.

The compiler generally aligns data on natural boundaries that are based on the target processor and the size of the data. Data is aligned on up to 4-byte boundaries on 32-bit processors, and 8-byte boundaries on 64-bit processors. In some cases, however, you can achieve performance improvements, or memory savings, by specifying a custom alignment for your data structures.

6.2.5 Type Qualifiers

6.2.5.1 Const

The const type qualifier is used to create constant variables. When a variable is created with const keyword, the value of that variable can't be changed once it is defined. That means once a value is assigned to a constant variable, that value is fixed and cannot be changed throughout the program. The default value of const variables is 0.

The keyword const is used at the time of variable declaration.

Syntax: **const datatype variableName ;**

Consider the Example: **test_const.c**

```
akshay@akshayv:Chapter6_Working_with_registers$ gcc test_const.c -o test_const
akshay@akshayv:Chapter6_Working_with_registers$ ./test_const
The default value of  Data : 0
The default value of  Temp1 : 0.000000
MaxVal: 255
Temp: 55.119999
Message: Bye World!!!
akshay@akshayv:Chapter6_Working_with_registers$
```

```c
#include <stdio.h>

const int Data;

int main()
{
    const int MaxVal = 255;
    const float Temp1;
    const float Temp = 55.12;
    const char Message[] = "Bye World!!!";

    printf("The default value of  Data : %d\r\n", Data);
    printf("The default value of  Temp1 : %f\r\n", Temp1);

    printf("MaxVal: %d\r\n", MaxVal);
    printf("Temp: %f\r\n", Temp);
    printf("Message: %s\r\n", Message);

    return 0;
}
```

6.2.5.2 Volatile

The volatile keyword is intended to prevent the compiler from applying any optimizations on variables that can change in ways that cannot be determined by the compiler. Variables declared as volatile are omitted from optimization because their values can be changed by code outside the scope of current code at any time. The system always reads the current value of a volatile object from the memory location rather than keeping its value in a temporary register at the point it is requested.

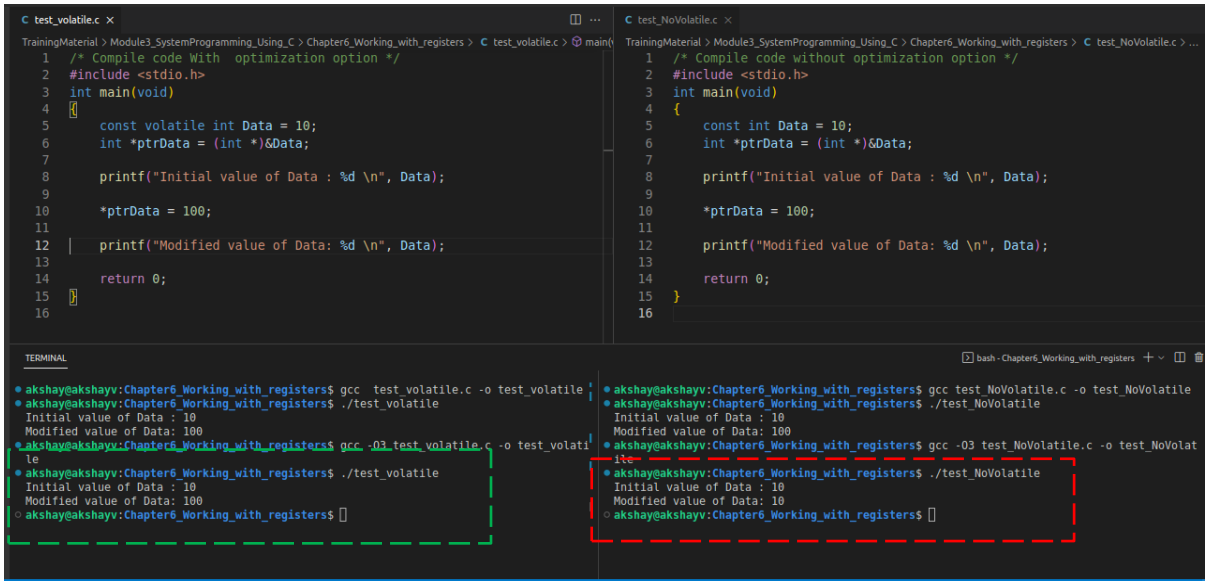
Syntax: **volatile datatype variableName;**

Optimization levels

If you use a higher optimization level for performance, then this has a higher impact on the other goals such as degraded debug experience, increased code size, and increased build time. If your optimization goal is code size reduction, then this has an impact on the other goals such as degraded debug experience, slower performance, and increased build time.

Optimization goal	Useful optimization levels
Smaller code size	-Oz
Faster performance	-O2, -O3, -Ofast, -Omax
Better debug experience	-O1
Better correlation between source code and generated code	-O0
Faster compile and build time	-O0
Balanced code size reduction and fast performance	-Os

For example, consider Program: **test_volatile.c** and **test_NoVolatile.c**



6.2.5.3 Restrict

Consider the example: **test_restrict.c**

It has two function which copy the array from source to destination.

We optimize the code with following settings -**std=c17 –O3**

We can output assembler code file using **-S** with **gcc**.

gcc -std=c17 -O3 test_restrict.c -S

From **test_restrict.s** check the asm code for both functions

```c
void copyArray(int n, int *restrict p, int *restrict q)
{
    while (n-- > 0)
    {
        *p++ = *q++;
    }
}

void FuncCopyArray(int n, int *p, int *q)
{
    while (n-- > 0)
    {
        *p++ = *q++;
    }
}
```

```
copyArray:
.LFB12:
    .cfi_startproc
    endbr64
    movslq  %edi, %rax
    movq    %rsi, %rdi
    movq    %rdx, %rsi
    testl   %eax, %eax
    jle .L1
    leaq    0(,%rax,4), %rdx
    jmp memcpy@PLT
    .p2align 4,,10
    .p2align 3
.L1:
    ret
    .cfi_endproc
.LFE12:
    .size   copyArray, .-copyArray
    .p2align 4
    .globl  FuncCopyArray
    .type   FuncCopyArray, @function
```

For **copyArray()** the code is optimized. But for **FuncCopyArray()**

It it less optimized.

```
FuncCopyArray:
.LFB13:
    .cfi_startproc
    endbr64
    movq    %rsi, %rcx
    leal    -1(%rdi), %r8d
    testl   %edi, %edi
    jle .L4
    leaq    4(%rdx), %rsi
    movq    %rcx, %rax
    subq    %rsi, %rax
    cmpq    $8, %rax
    jbe .L6
    cmpl    $2, %r8d
    jbe .L6
    movl    %edi, %esi
    xorl    %eax, %eax
    shrl    $2, %esi
    salq    $4, %rsi
    .p2align 4,,10
    .p2align 3
.L7:
    movdqu  (%rdx,%rax), %xmm0
    movups  %xmm0, (%rcx,%rax)
    addq    $16, %rax
    cmpq    %rax, %rsi
    jne .L7
    movl    %edi, %esi
    andl    $-4, %esi
    movl    %esi, %eax
    subl    %esi, %r8d
    salq    $2, %rax
    addq    %rax, %rcx
    addq    %rdx, %rax
    cmpl    %esi, %edi
    je  .L4
    movl    (%rax), %edx
    movl    %edx, (%rcx)
```

7. Context Management

Unit 7.1 – Bringing up CPU

Unit 7.2 – Requirements of C startup

Unit 7.3 – What is Execution Context?

Unit 7.4 – Interrupt Context

Key Learning Outcomes

At the end of this module, you will be able to:

1. Explain the startup process of the CPU.
2. Explain the C startup requirements, such as code, data, BSS, stack, and heap segments;
3. setting up constants; zeroing BSS; and constructors and destroyers. the environment of execution context.
4. Explain the environment of interrupt.

UNIT 7.1: Bringing up CPU

Unit Objectives

At the end of this unit, you will be able to:

1. Explain the startup process of the CPU.

7.1.1 Startup Sequence

When we reboot our computer, it must start up again, initially without any notion of an operating system. Somehow, it must load the operating system. Whatever variant that may be from some permanent storage device that is currently attached to the computer (e.g. a hard disk, a USB dongle, SD card, eMMC, etc.). The CPU starts and fetches instructions into RAM from the BIOS, which is stored in the ROM.

7.1.2 BIOS=Basic Input Output System

Firmware code lives on flash memory. On the motherboard. It is the minimum software that system needs. Designed to run as the first code by PC. Identify, test, and initialize system devices. It helps in the functioning of all the input/output devices.

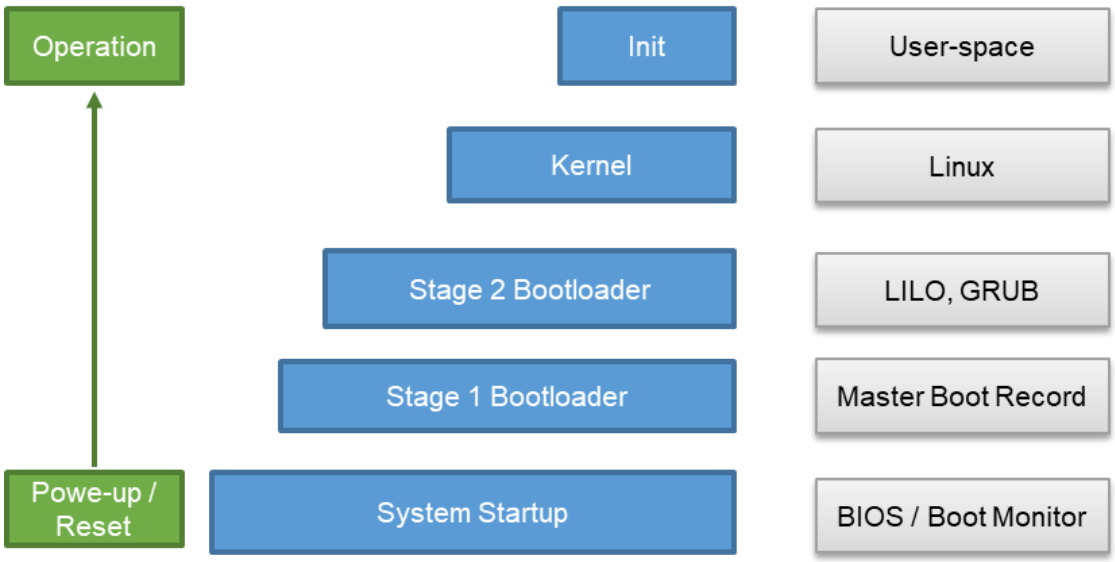


Fig 7.1.2 Devices

7.1.3 Functions of BIOS

POST (Power on Self-Test):

The Power on Self-Test happens each time you turn your computer on. Your computer does so much when it's turned on, and this is just part of that. It initializes the various hardware devices. It is an important process to ensure that all the devices operate smoothly without any conflicts.

BIOSes create tables describing the devices in the computer. The POST first checks the bios and then tests the CMOS RAM. If there is no problem with this then POST continues to check the CPU, hardware devices such as the Video Card, and the secondary storage devices such as the Hard Drive, Floppy Drives, Zip Drive, or CD/DVD Drives. If some errors are found then an error message is displayed on the screen or a number of beeps are heard. These beeps are known as POST beep codes.

7.1.4 Master Boot Record

The Master Boot Record (MBR) is a special boot sector at the beginning of the disk. The MBR contains the code that loads the rest of OS, known as bootloader. Along with the bootloader program, MBR also contains details regarding the partitions of the hard disk. The size of the MBR is commonly less than or equal to 512 bytes.

The MBR mainly consists of 3 parts:

Master Boot Code

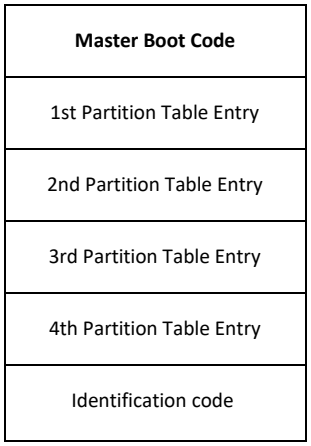
The MBR begins with the master boot routine that contains a variable loader code. Users can boot various operating systems from the MBR. Once the hard disk is booted, MBR passes control to the Operating System that has been registered in the partition table.

Disk Partition Table (DPT)

The disk partition table is located at the first sector of each hard disk and contains locations of the partitions. The disk partition table is usually 64 bytes long. It contains a maximum of 4 partitions that can be 16 bytes each.

Identification Code

An identification code is used to identify an MBR and acts as a closing signature. Its value is AA55 H and may also be written as 55AA H. The identification code is 2 bytes long.



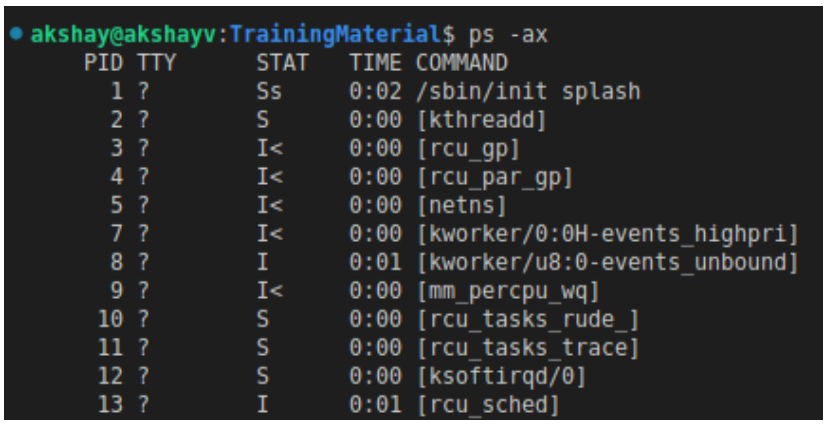
7.1.5 Bootloader of Linux

GNU GRUB

GRUB stands for GRand Unified Bootloader. Its function is to take over from BIOS at boot time, load itself, load the Linux kernel into memory, and then turn over execution to the kernel. Once the kernel takes over, GRUB has done its job, and it is no longer needed. The GRUB splash screen is often the first thing you see when you boot your computer. It has a simple menu where you can select some options. If you have multiple kernel images installed, you can use your keyboard to select the one you want your system to boot with. By default, the latest kernel image is selected.

Kernel

The kernel is often referred to as the core of any operating system, Linux included. It has complete control over everything in your system. In this stage of the boot process, the kernel that was selected by GRUB first mounts the root file system that's specified in the **grub.conf** file. Then it executes the **/sbin/init** program, which is always the first program to be executed. You can confirm this with its process id (PID), which should always be 1. The kernel then establishes a temporary root file system using the Initial RAM Disk (initrd) until the real file system is mounted.



Init

At this point, your system executes run-level programs. At one point it would look for an init file, usually found at /etc/inittab to decide the Linux run level. Modern Linux systems use systemd to choose a run level instead. A run-level is used to decide the initial state of the operating system.

These are the available run levels:

Level 0: System Halt.

Level 1: Single user mode.

Level 2: Full multiuser mode without network.

Level 3: Full multiuser mode with network.

Level 4: user definable.

Level 5: Full multiuser mode with network and X display manager.

Level 6: Reboot.

By default, most of the LINUX based system boots to runlevel 3 or runlevel 5.

In addition to the standard runlevels, users can modify the preset runlevels or even create new ones according to the requirement.

UNIT 7.2: Requirements of C startup

Unit Objectives

At the end of this unit, you will be able to:

1. Explain the C start-up requirements, such as code, data, BSS, stack, and heap segments;setting up constants; zeroing BSS; and constructors and destroyers.

7.2.1 C Startup Code

Int Bare metal system it is not possible to directly execute C code main(), when the processor comes out of reset. C programs need some basic pre-requisites to be satisfied. Before transferring control to C code, the following have to be setup correctly:

- Stack
- Global variables
    - Initialized
    - Uninitialized
    - Read-only data

The CPU begins to execute code at a determined address, that could be 0x0 the initial value of the Stack Pointer. Startup code runs before the main () function starts its execution.

What is actually needed to start the execution of the main function?

All uninitialized variables are zero. These are stored in the bss section of the final elf file. All initialized variables are actually initialized. These are stored in the .data section of the final elf file. All static objects are initialized, they may need to get their constructors called if they are not trivial. Function pointers to these static initialization routines are stored in the init_array section. The stack pointer is correctly set during startup. Some other machine dependent features like enabling access to the floating-point coprocessor (VFP coprocessor on most ARM microcontroller architectures, etc.).

Default startup code is given in the assembly language for the target processor. It makes sense, however, reimplementing this code in C, for the purposes of creating a more generic code that can be used for multiple devices.

Vector table for the NVIC (Nested Vectored Interrupt Controller):

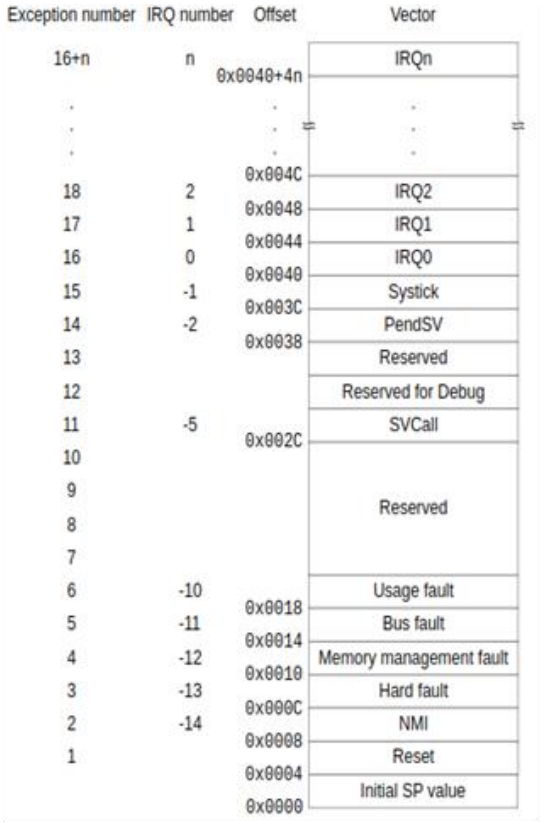
Define the vector table for the NVIC.

Upon an exception or interrupt it looks up the address of the corresponding ISR.

This table contains:

- The stack pointer and program counter initial value.
- The reset vectors
- All exception vectors
- All external interrupt vectors.
- Fault handlers.

When a system reset occurs, execution starts from the reset vector.
The processor loads the value of the MSP (main stack pointer) with the highest ram address (defined by the linker as _Stack Top).
The __StackTop variable is actually defined in the linker script.
_sram_stacktop = ORIGIN(SRAM) + LENGTH(SRAM);



Credits: https://documentation-service.arm.com/static/5ea823e69931941038df1b02?token=

Vector table for the NVIC (Nested Vectored Interrupt Controller):

- The vector table contains the reset value of the stack pointer, and the start addresses, also called exception vectors, for all exception handlers.
- The processor handles exceptions using: Interrupt Service Routines (ISRs): The IRQ interrupts are the exceptions handled by ISRs. Fault handlers: Hard Fault, MemManage fault, Usage Fault, and Bus Fault are fault exceptions handled by the fault handlers. System handlers: NMI, PendSV, SVCall SysTick, and the fault exceptions are all system exceptions that are handled by system handlers.

```c
/* Exception Table */
__attribute__ ((section(".vectors"), used))
void (* const  _exceptions[CORTEX_M3_EXCEPTIONS])(void) = {
    (void (*)(void))((uint32_t)&_sram_stacktop),          // 00: Reset value of the Main Stack Pointer
    _Reset_Handler,                                       // 01: Reset value of the Program Counter
    _NMI_Handler,                                         // 02: Non-Maskable Interrupt (NMI)
    _Hard_Fault_Handler,                                  // 03: Hard Fault
    _Memory_Mgmt_Handler,                                 // 04: Memory Management Fault
    _Bus_Fault_Handler,                                   // 05: Bus Fault
    _Usage_Fault_Handler,                                 // 06: Usage Fault
    _Unused_Handler,                                      // 07: --
    _Unused_Handler,                                      // 08: --
    _Unused_Handler,                                      // 09: --
    _Unused_Handler,                                      // 10: --
    _SVCall_Handler,                                      // 11: Supervisor Call
    _Debug_Monitor_Handler,                               // 12: Debug Monitor
    _Unused_Handler,                                      // 13: --
    _PendSV_Handler,                                      // 14: Pendable req serv
    _SysTick_Handler,                                     // 15: System timer tick
};
```

Initialize the .data section:

This section is defined in the linker script with different VMA (Virtual Address) and LMA (Load Address). Since it has to be loaded to Flash, but used from RAM when the execution starts. To make sure that all C code can use the initialized data within the .data section, it has to be copied over from Flash to RAM by the startup code. To accomplish this, it uses the following symbols defined by the linker: _flash_sdata, _sram_sdata and _sram_edata.

Initialize the. bss section to zero:

It uses the symbols _sram_sbss and _sram_ebss to obtain the address range that should be set to zero. These symbols are defined in the linker script.

_Reset_Handler:

When a reset occurs, it will start executing the code at the address indicated by the _Reset_Handler function pointer.

This is where we setup and call main ().

We'll create a separate section. startup so this resides immediately after the vector table

Initialize the. Data section

Initialize the. bss section to zero

_Reset_Handler

```c
__attribute__ ((section(".startup")))
void _Reset_Handler(void)
{
    /* Copy the data segment from flash to sram */
    uint32_t *pSrc = &_flash_sdata;
    uint32_t *pDest = &_sram_sdata;

    while(pDest < &_sram_edata)
    {
        *pDest = *pSrc;
        pDest++;
        pSrc++;
    }

    /* Zero initialize the bss segment in sram */
    pDest = &_sram_sbss;

    while(pDest < &_sram_ebss)
    {
        *pDest = 0;
        pDest++;
    }

    /* Call main() */
    main();

  /* main() isn't supposed return
   * - if it does, we need to identify this
   * for now, we'll loop infintely
   */
    while(1);
}
```

UNIT 7.3: What is Execution Context?

Unit Objectives

At the end of this unit, you will be able to:

1. Explain the execution context's environment.

7.3.1 Execution Context

Execution is the process by which a computer reads and acts on the instructions of a computer program. The context in which execution takes place is crucial. Very few programs execute on a bare machine. Programs usually contain implicit and explicit assumptions about resources available at the time of execution. In order for programs and interrupt handlers to work without interference and share the same hardware memory and access to the I/O system. In a multitasking operating system running on a digital system with a single CPU/MCU it is required to have some sort of software and hardware facilities to keep track of an executing processes data. It includes memory page addresses, registers etc and to save and recover them back to the state they were in before they were suspended. This is achieved by a context switching. In Linux-based operating systems, a set of data stored in registers is usually saved into a process descriptor in memory to implement switching of context. An execution context consists of a heap, a stack, code, an instruction pointer, registers etc. When you're programming in C you only use the heap explicitly. The stack etc. are implicitly used when doing procedure call sand declaring local variables.

Context Switching

Context Switching involves storing the context or state of a process so that it can be reloaded when required and execution can be resumed from the same point as earlier. This is a feature of a multitasking operating system and allows a single CPU to be shared by multiple processes. When switching is performed in the system, the old running process's status is stored as registers, and the CPU is assigned to a new process for the execution of its tasks. While new processes are running in a system, the previous ones must wait in the ready queue. The old process's execution begins at that particular point at which another process happened to stop it.

Context Switching Triggers: Here are the triggers that lead to context switches in a system:

Interrupts:

The CPU requests the data to be read from a disk.

In case there are interrupts, the context switching would automatically switch a part of the hardware that needs less time to handle the interrupts.

Multitasking:

Context switching is the characteristic of multitasking.

They allow a process to switch from the CPU to allow another process to run.

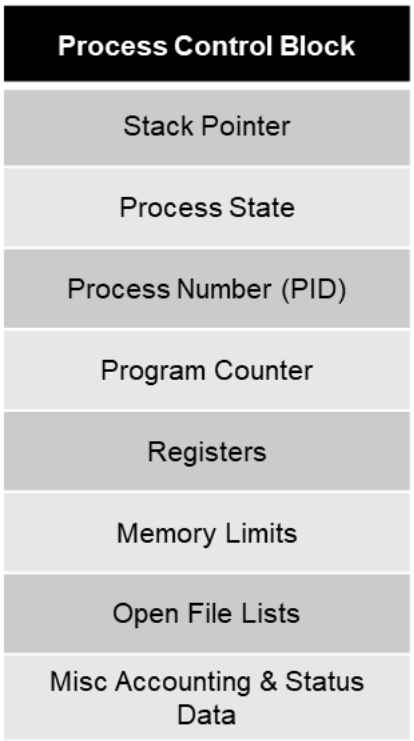
When switching a given process, the old state gets saved so as to resume the execution of the process at the very same point in a system.

Kernel/User Switch:

It's used in the OS when it is switching between the kernel mode and the user mode.

Process Table and Process Control Block (PCB)

While creating a process the operating system performs several operations. To identify the processes, it assigns a process identification number (PID) to each process. As the operating system supports multi-programming, it needs to keep track of all the processes. For this task, the process control block (PCB) is used to track the process's execution status. Each block of memory contains information about the process state, program counter, stack pointer, status of opened files, scheduling algorithms, etc. All these information is required and must be saved when the process is switched from one state to another. When the process makes a transition from one state to another, the operating system must update information in the process's PCB. The operating system maintains pointers to each process's PCB in a process table so that it can access the PCB quickly.



Manage execution contexts:

We will explore the execution context by means of library **ucontext.**

- The **ucontext.h** header file defines the **ucontext_t** type as a structure that can be used to store the execution context of a user-level thread in user space. Structure ucontext_t holds a pointer to an execution stack, copy if registers and instruction pointer.

- The same header file also defines a small set of functions used to manipulate execution contexts. Read the following manual pages:
  - ucontext.h
  - getcontext()
  - setcontext()
  - makecontext()
  - swapcontext()
- Using these functions we can capture the current execution context, change it and even start using another context. We can switch from one execution to another by context switching. The header files might not be installed on the system so install the header files:
- s**udo apt-get install linux-headers-$(uname -r)**

The first lets try to get hold of current execution context and swap one context for another.
Consider the example program: **switch.c**

- This example saves the context in main with the getcontext() statement.
- It then returns to that statement from the function func using the setcontext() statement.
- Since getcontext() always returns 0 if successful, the program uses the variable x to determine if getcontext() returns as a result of setcontext() or not.

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

int x = 0;
ucontext_t context, *cp = &context;
void func(void)
{
    x++;// Incr the x value

    setcontext(cp); // set context to pointed by cp
    // The function never comes here because we switched back to context pointed by cp
    printf("How did i reach here!!!!\r\n");
}

int main(void)
{
    // saves the context of main with the getcontext() statement.
    getcontext(cp);
    if (!x){
        printf("[%s:%d]\t[Value X = %d]\tgetcontext has been called\r\n",__FUNCTION__,__LINE__,x);
        func();
    }
    else{
        printf("[%s:%d]\t[Value X = %d]\tsetcontext has been called\r\n", __FUNCTION__, __LINE__,x);
    }
    printf("[%s:%d]\tFinnaly Exiting!!!\r\n",__FUNCTION__);
    return 0;
}
```

```
akshay@akshayv:ExecutionContext$ gcc switch.c -o switch
akshay@akshayv:ExecutionContext$ ./switch
[main:21]       [Value X = 0]   getcontext has been called
[main:25]       [Value X = 1]   setcontext has been called
[main:27]       Finnaly Exiting!!!
akshay@akshayv:ExecutionContext$
```

Consider the Program: context_switch.c
Program demonstrate how to create and manipulate execution contexts.
Study the source.
The program handover context to functions foo () and bar() using the swapcontext() function.
This context handover process will continue based on the number of iterations defined in #define N.

```
akshay@akshayv:ExecutionContext$ gcc context_switch.c -o context_switch
akshay@akshayv:ExecutionContext$ ./context_switch
foo (0)
bar (0)
foo (1)
bar (1)
foo (2)
bar (2)
foo (3)
bar (3)
Exiting The context foo()
foo  -  done!
akshay@akshayv:ExecutionContext$
```

Multithreading in C

Thread

Is a free unit of execution made inside the context of a process or application that is executed. At the point when different threads are executing in a process simultaneously, then it is called as multithreading. The threads are unaware of other threads in a process. Threads share same global memory (data and heap segments), but each thread has its own stack (automatic variables).

Multithreading

It allows multiple threads to be created inside a process, executing freely but concurrently sharing the process resources. Typical process resources are dynamically allocated memory, network connections, open files etc. Based on hardware, threads can execute in parallel if they are divided to their own CPU core. C does not have built-in support for multithreading, instead it relies on the OS. In Linux OS we have POSIX threads.

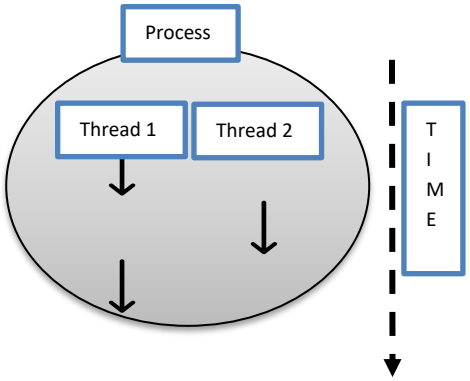


Fig 7.3.1 Multithreading

POSIX threads:

Specifies a set of interfaces (functions, header files) for threaded programming commonly known as POSIX threads, or Pthreads. The interfaces provided by the Pthread API can be grouped as shown in chart:

Some functions which we will use in demo app:

pthread_create() -: function starts a new thread in the calling process.

It takes 4 arguments:

pthread_t: pointer to thread id, it is an integer used to identify the thread in the system.

attr : Used to set the thread attributes. We can specify the thread attribute's or NULL for default value.

start_routine: C function that the thread will execute

arg: argument that can be passed to start_routine.

pthread_join() -: function waits for the thread specified by t*hread* to terminate. If that thread has already terminated, then pthread_join() returns immediately.

Takes two arguments:

pthread_t

retval: pointer to the status argument passed by the thread as a part of pthread_exit().

Routine Prefix	Functional Group
pthread_	Thread management API's
pthread_attr_	Thread attributes objects
pthread_mutex_	Mutex that deal with synchronization
pthread_mutexattr_	Mutex attributes objects
pthread_cond_	Condition variables
pthread_condattr_	Condition attributes objects
pthread_key_	Thread-specific data keys
pthread_rwlock_	Read/write locks
pthread_barrier_	Synchronization barriers

Mutex API:

Mutex – Mutual Exclusion Object. It is a locking mechanism used to synchronize the access to the CPU / Shared resource. It is created so that multiple thread can take turns sharing the CPU resource, Global Variables or access to file etc. When the thread wants to access the resource then they can call the mutex lock and proceed with the task. At any time only one thread can acquire the Mutex.

- pthread_mutex_init() -: function initializes the mutex referenced by mutex with attributes specified by attr. If NULL, the default mutex attributes are used.
- pthread_mutex_lock() -: Mutex can be locked using this function.
- pthread_mutex_unlock() -: Mutex can be released using this function.
- To compile the code with GCC for multithreading which use POSIX pthread.h:

We use -lpthread -: the linker find those symbols in the pthread library during the linking stage.

POSIX threads:

Consider example program: test_multithreading.c

```c
#include <pthread.h>
#include <unistd.h>
#include <stdio.h>

// Mutex for lock on the variable or a shared resourcec:
pthread_mutex_t lock;
// Global variable :
// Every process should have the exclusive access
volatile int ProcCount;

void test_proc()
{
    // Current Thread ID:
    printf("I am from Thread: 0x%0X\r\n",(int)pthread_self());
    sleep(2);
    // Lock the Mutex:
    pthread_mutex_lock(&lock);
    int i = 0;

    // Shared variable between all the processes:
    // This Variable is incremented by each process.
    ProcCount++;

    // Some tasks:
    while (i < 5)
    {
        printf("%d", ProcCount);
        sleep(1);
        i++;
    }
    printf("...Done : 0x%0X\r\n", (int)pthread_self());
    // Unlock the mutex:
    pthread_mutex_unlock(&lock);
}
```

```c
int main(void)
{
    int err;
    pthread_t t1, t2, t3;
    int ProcessNum = 0;

    // Mutex init:
    if (pthread_mutex_init(&lock, NULL) != 0)
    {
        printf("Mutex initialization failed.\n");
        return 1;
    }

    // Init the variable to 0
    ProcCount = 0;

    // Create 3 threads which execute the test_proc()
    pthread_create(&t1, NULL, (void*)test_proc, NULL);
    pthread_create(&t2, NULL, (void*)test_proc, NULL);
    pthread_create(&t3, NULL, (void*)test_proc, NULL);

    // Wait for a thread to terminate, detaches the thread,
    // Finnaly returns the threads exit status
    pthread_join(t1, NULL);
    pthread_join(t2, NULL);
    pthread_join(t3, NULL);

    return 0;
}
```

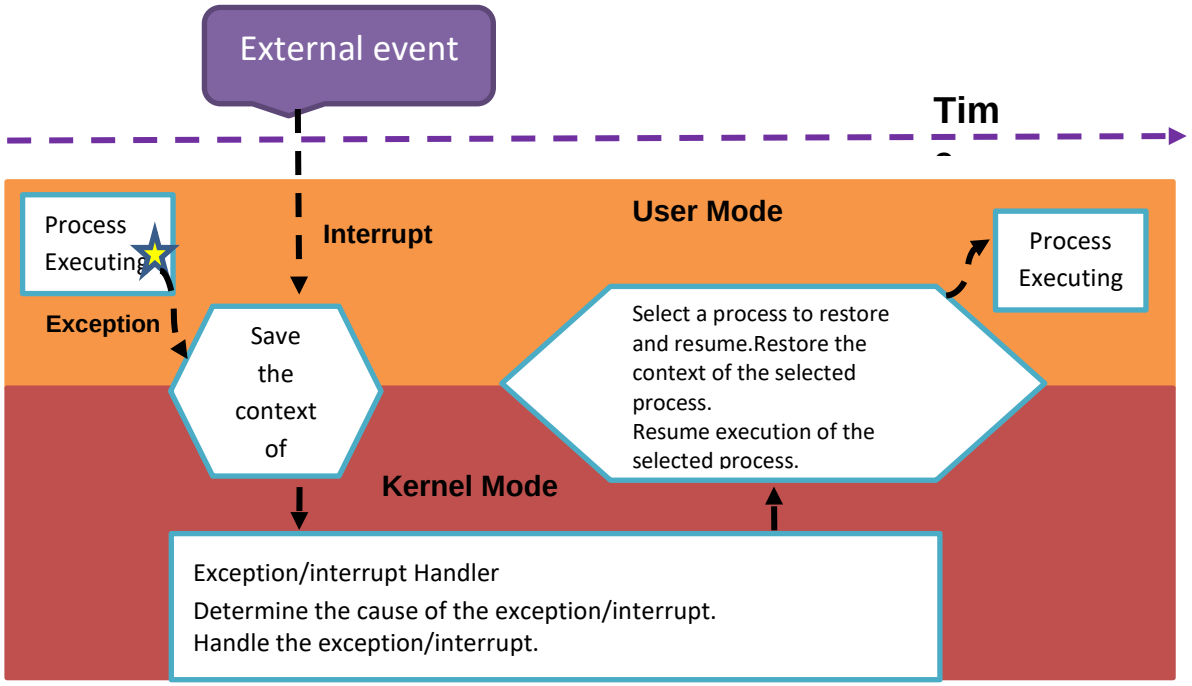
Output Observation:

- You can see we created three threads.
- Each thread executes the function **test_proc()**.
- As the mutex is locked after entering the function each waits untill the mutex is released.
- So this is how each thread can execute and operate on the global variable **ProcCount** exclusively.

```
akshay@akshayv:Chapter7_Context_Management$ gcc test_multithreading.c -lpthread -o test_multithreading
akshay@akshayv:Chapter7_Context_Management$ ./test_multithreading
 I am from Thread: 0xA2DF3640
 I am from Thread: 0xA1DF1640
 I am from Thread: 0xA25F2640
 11111...Done : 0xA1DF1640
 22222...Done : 0xA2DF3640
 33333...Done : 0xA25F2640
akshay@akshayv:Chapter7_Context_Management$
```

7.3.2 Interrupt Context

When an exception or interrupt occurs, execution transition from user mode, to kernel mode where the exception or interrupt is handled. When the exception or interrupt has been handled, execution resumes in user space. In detail, the following steps must be taken to handle an exception or interrupts.



Saving context

The exception or interrupt handler uses the same CPU as the currently executing process. When entering the exception or interrupt handler, the values in all CPU registers to be used by the exception or interrupt handler must be saved to memory. The saved register values can later be restored before resuming execution of the process.

Determine the cause

The handler may have been invoked for a number of reasons. The handler thus needs to determine the cause of the exception or interrupt. Information about what caused the exception or interrupt can be stored in dedicated registers or at predefined addresses in memory.

Handle the exception/interrupt

Next, the exception or interrupt needs to be serviced. For instance, if it was a keyboard interrupt, then the key code of the keypress is obtained and stored somewhere, or some other appropriate action is taken. If it was an arithmetic overflow exception, an error message may be printed or the program may be terminated.

Interrupt dispatch

There are two different handlers, __irq_usr and __irq_svc..The handlers loop around this code until no interrupts remain. If there is an interrupt, the code will branch to do_IRQ that exists in arch or arm or kernel or irq.c. At this point, the code is the same in all architectures and you call an appropriate handler written in C.

Select a process to resume

The exception or interrupt have now been handled and the kernel.

The kernel may choose to resume the same process that was executing prior to handling the exception or interrupt or resume execution of any other process currently in memory.

Restoring context

The context of the CPU can now be restored for the chosen process by reading and restoring all register values from memory.

Resume

The process selected to be resumed must be resumed at the same point it was stopped. The address of this instruction was saved by the machine when the interrupt occurred, so it is simply a matter of getting this address and make the CPU continue to execute at this address.

7.3.3 Processor Modes

Many processers run more than a single process simultaneously. In a multi-process system, each process gets control of the processor and its memory, ports, and I/O devices for a limited “slice” of time. At the end of its allotted time slice, each process must surrender control and wait idly for its scheduled time-slice to come around again. Because the processor and its resources can be shared by several active processes, no single process can be allowed to compromise the others. This means that some system resources must be protected, so that processes can’t interfere with one another. No user process should be able to reset the processor or put it in a non-responsive state, and no process should be able to change system settings (like exception vectors) that would affect other processes. To help manage such situations, most processors have different operating modes that allow different levels of “privileged” access to areas of memory, system settings, peripheral devices, and other resources.

Consider ARM processors that support different processor modes, depending on the architecture version:

Processor mode	Description
User	The basic mode in which application programs run. User mode is the only unprivileged mode, and it has restricted access to system resources. Typically, a processor spends more than 99% of its time in user mode.
FIQ - Fast Interrupt Request	Is entered in response to a fast interrupt request from an external device. It is used to provide faster service for more urgent requests.
IRQ - Interrupt Request	Is entered in response to a normal interrupt request from an external device.
Supervisor	Provides unrestricted access to all system resources. Supervisor mode is entered on reset or power-up, or when software executes a Supervisor Call instruction (SVC).
Abort	Is entered if a program attempts to access a non-existing memory location. Abort mode also offers access to a few private registers that other modes can't access.
Undefined	Is entered for any instruction-related exceptions, including any attempt to execute an unimplemented instruction.
System	provides unrestricted access to all system resources. Is typically only entered when required to manage a particular resource.
Monitor	Is available in some implementations to change between secure and non-secure states, and for debugging.

7.3.3.1 Registers

ARM processors have 37 registers.

The following registers are available:

- Thirty general-purpose, 32-bit registers
- The Program Counter (PC)
- The Application Program Status Register (APSR)
- Saved Program Status Registers (SPSRs).

Of those, 20 registers are hidden from a program at different times. These are called banked registers. The shaded blocks in figure 7.3.3.1 are banked registers. The registers are arranged in partially overlapping banks. There is a different register bank for each processor mode. The banked registers give rapid context switching for dealing with processor exceptions and privileged operations.

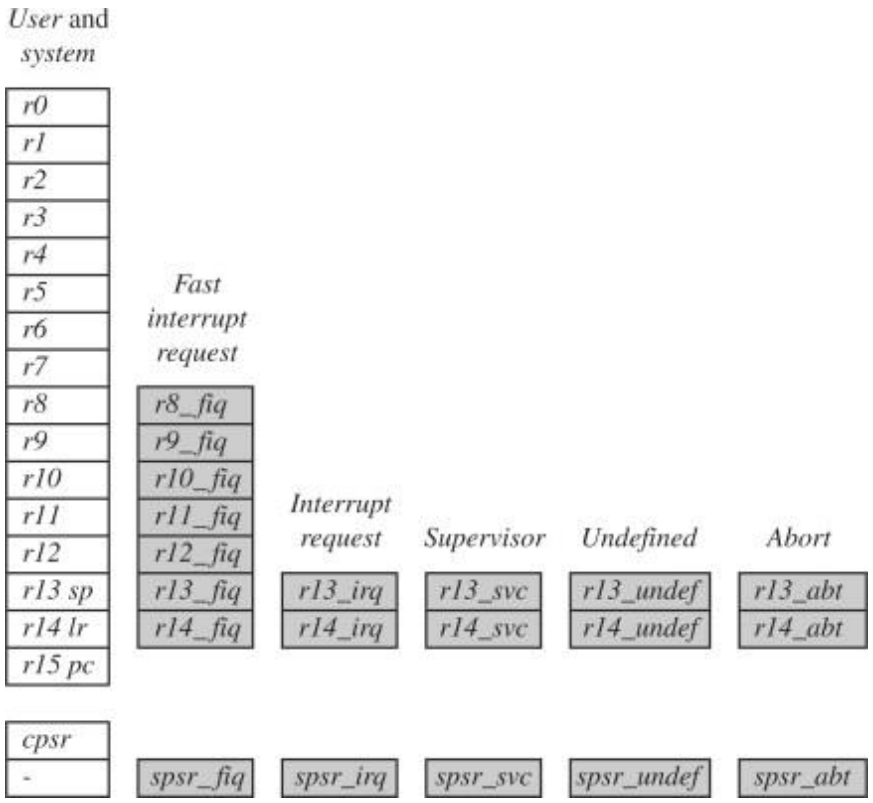


Fig 7.3.3.1 Registers

Fifteen general-purpose registers are visible at any one time, depending on the current processor mode. These are r0-r12, sp (Stack Pointer), lr (Link Register).

SP

Sp (or r13) is the *stack pointer*. The C and C++ compilers always use sp as the stack pointer. In Thumb-2 instruction set, sp is strictly defined as the stack pointer, so many instructions that are not useful for stack manipulation are unpredictable if they use sp. Use of sp as a general-purpose register is discouraged.

LR

In User mode, lr (or r14) is used as a *link register* to store the return address when a subroutine call is made. It can also be used as a general-purpose register if the return address is stored on the stack. In the exception handling modes, lr holds the return address for the exception, or a subroutine return address if subroutine calls are executed within an exception. lr can be used as a general-purpose register if the return address is stored on the stack.

Program Counter (PC)

The Program Counter is accessed as pc (or r15). It is **incremented by one word (four bytes)** for each instruction in ARM state, or by the size of the instruction executed in Thumb state. Branch instructions load the destination address into pc. You can also load the PC directly using data operation instructions. For example, to return from a subroutine, you can copy the link register into the PC using:

Application Program Status Register (APSR)

The APSR holds copies of the Arithmetic Logic Unit (ALU) status flags. They are used to determine whether conditional instructions are executed or not.

Current Program Status Register (CPSR)

The CPSR holds:

- The APSR flags
- The current processor mode
- Interrupt disable flags
- Current processor state (ARM, Thumb, ThumbEE, or Jazelle)

Execution state bits for the IT (If-Then) instruction block.

The execution state bits control conditional execution in the IT block and are only available on ARMv6T2 and above. Only the APSR flags are accessible in all modes. The remaining bits of the CPSR are accessible only in privileged modes, using MSR and MRS instructions.

Saved Program Status Registers (SPSRs)

The SPSRs are used to store the CPSR when an exception is taken. One SPSR is accessible in each of the exception-handling modes. User mode and ystem mode do not have an SPSR because they are not exception handling modes.

For example, when the processor is in the interrupt request mode. The instructions you execute still access registers named r13 and r14. However, these registers are the banked registers r13_irq and r14_irq. The user mode registers r13_usr and r14_usr are not affected by the instruction referencing these registers. A program still has normal access to the other registers r0 to r12. Figure 7.3.3.1 illustrates what happens when an interrupt forces a mode change. The figure 7.3.3.1 shows the core changing from user mode to interrupt request mode, which happens when an interrupt request occurs due to an external device raising an interrupt to the processor core. This change causes user registers r13 and r14 to be banked. The user registers are replaced with registers r13_irq and r14_irq, respectively.

Note r14_irq contains the return address and r13_irq contains the stack pointer for interrupt request mode. Fig also shows new register appearing in interrupt request mode: the saved program status register (spsr), which stores the previous mode's cpsr. You can see in the diagram the cpsr being copied into spsr_irq. To return back to user mode, a special return instruction is used that instructs the core to restore the original cpsr from the spsr_irq and bank in the user registers r13 and r14.

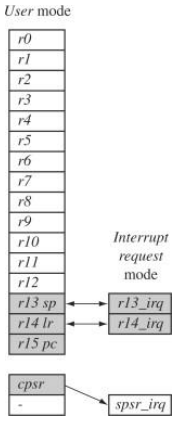


Fig 7.3.3.1 Saved Program Status Register

8. Linux Fundamentals

Key Learning Outcomes

At the end of this unit, you will be able to:

1. Explain the architecture of Linux
2. Explain about the kernel and its types
3. Discuss the file system of Linux
4. Elucidate the files/devices invoked in Linux
5. Analyze the code using debug commands
6. Explain the kernel debug mode

UNIT 8.1: Linux Kernel Basics

Unit Objectives

At the end of this unit, you will be able to:

1. Explain the startup process of the CPU
2. Explain the architecture of Linux
3. Explain about the kernel and its types
4. Discuss the kernel system calls
5. Develop a program using socket to establish the communication between two systems

8.1.1 Linux Introduction

What Is Linux?

Just like Windows, iOS, and Mac OS, Linux is an operating system. The most popular platforms on the planet, Android, is powered by the Linux operating system. The operating system manages the communication between your software and your hardware. Linux has gained in popularity over the years due to it being open source. Hence, based on a UNIX like design, and ported to more platforms compared to other competing operating systems. Linux is deployed on a wide variety of computing systems, such as embedded devices, mobile devices, personal computers, servers, mainframes, and supercomputers.

Linux or GNU/Linux?

Linux as an operating system is referred to in some cases as "Linux" and in others as "GNU/Linux." Linux is the kernel of an operating system. The wide range of applications that make the operating system useful are the GNU software. For example, the windowing system, compiler, variety of shells, development tools, editors, utilities, and other applications exist outside of the kernel, many of which are GNU software. That's the reason many consider "GNU Linux" as appropriate name.

8.1.2 Linux Architecture

Figure 8.1.2 shows fundamental architecture of Linux OS.

User or Application Space:

This is where the user applications are executed.

GNU C Library (glibc):

Provides the core libraries for the GNU system and GNU/Linux systems, as well as many other systems that use Linux as the kernel. Provides the system call interface that connects to the kernel. Provides the mechanism to transition between the user-space application and the kernel. This is important because the kernel and user application occupy different protected address spaces. Each user-space process occupies its own virtual address space. The kernel occupies a single address space.

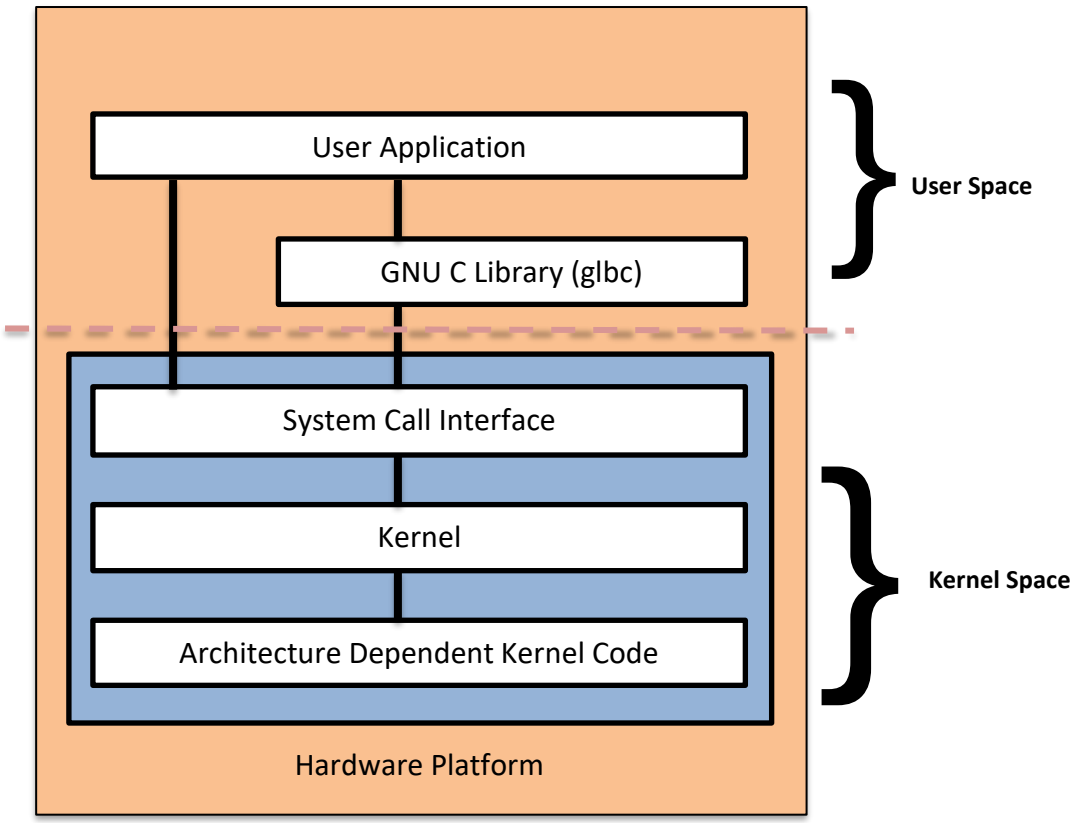


Fig 8.1.2 Linux Architecture

Kernel Space - Kernel

The kernel code, which can be more accurately defined as the architecture-independent kernel code. This code is common to all of the processor architectures supported by Linux. Kernel is really nothing more than a resource manager. Resource being managed is a process, memory, or hardware device, the kernel manages and arbitrates access to the resource between multiple users in both kernel and user space.

Kernel Space - Architecture Dependent Kernel Code

The architecture-dependent code, which forms what is more commonly called a BSP (Board Support Package). This code serves as the processor and platform-specific code for the given architecture. The Linux kernel are fairly portable across platforms. Which means that most of the code runs on all the supported architectures without the need to differentiate among them.

8.1.3 The Linux Kernel

A kernel represents the core aspect of the Linux distributions for desktop computers and servers. It has a monolithic architecture, and the operating system operates entirely in the kernel space. Linux kernel works as a layer between the software and hardware of a device. It contains many device drivers that create a communication interface between the hardware and software of a device. The kernel consists of various modules that can dynamically be loaded and unloaded. A module can be configured as built-in or loadable. This kind of architecture would extend the capabilities of the OS

and allows easy extensions to the kernel. It eliminates the tedious task of bringing down and recompiling the whole kernel for little changes.

Advantages of using Kernel Modules:

- For small changes/bug fixing we don't have to rebuild kernel. This saves time and prevents introducing errors in rebuilding and reinstalling the base kernel.
- It is easier to diagnose system problems.
- A bug in a device driver which is bound to kernel can stop the system from booting.
- And it can be hard to tell which part of base kernel caused trouble.
- If the same device driver is a module, though, the base kernel is up and running before the device driver even gets loaded.
- If the system breaks after the kernel is up and running, it's an easy matter to track the problem down to the trouble-making device driver and just not load that device driver until the problem is fixed.
- Using modules can save memory, because they are loaded only when the system is actually using them.
- Modules are much faster to maintain and debug.
- It is possible to try out different parameters or even change the code repeatedly in rapid succession, without waiting for a boot.

Loadable Kernel Modules:

In computing, a loadable kernel module (or LKM) is an object file that contains code to extend the running kernel, or so-called base kernel, of an operating system. LKMs are typically used to add support for new hardware (as device drivers) and/or filesystems, or for adding system calls. When the functionality provided by an LKM is no longer required, it can be unloaded in order to free memory and other resources. If you want to add code to the Linux kernel, the first thing you need to do is to add some source files to the kernel source tree. There may be situations where you are required to add code to the kernels while it is running, this process is called a loadable kernel module. Without loadable kernel modules, an operating system would have to include all possible anticipated functionality compiled directly into the base kernel. Much of that functionality would reside in memory without being used, wasting memory, and would require that users rebuild and reboot the base kernel every time they require new functionality.

Module Location:

Find the information about the system:

- We can use command un name to print the system information.

```
pi@raspberrypi:~ $ uname -a
Linux raspberrypi 5.15.61-v7+ #1579 SMP Fri Aug 26 11:10:59 BST 2022 armv7l GNU/Linux
pi@raspberrypi:~ $ uname -r
5.15.61-v7+
pi@raspberrypi:~ $
```

Fig 8.1.3 Module location

Uname - a: Prints all the system information in the following order: Kernel name, network node hostname, kernel release date, kernel version, machine hardware name, hardware platform, operating system

Uname - r: Prints print the kernel release

The code necessary to create a new kernel with new module included, or old modules removed is usually:

/lib/modules/$(uname -r)/kernel

On some distributions the code is found:

/usr/lib/modules/$(uname -r)/kernel

The /usr/lib and /lib directory are where Linux stores object libraries and shared libraries that are necessary to run certain commands, including kernel code.

Kernel Modules Subdirectories:

```
pi@raspberrypi:/lib/modules/5.15.61-v7+/kernel $ cd /lib/modules/$(uname -r)/kernel
pi@raspberrypi:/lib/modules/5.15.61-v7+/kernel $ tree -L 1
.
├── arch
├── crypto
├── drivers
├── fs
├── kernel
├── lib
├── mm
├── net
└── sound

9 directories, 0 files
pi@raspberrypi:/lib/modules/5.15.61-v7+/kernel $
```

Fig 8.1.3 Kernel Module subdirectories

Directory	Description
arch	The arch contains all of the architecture specific kernel code.
mm	This directory contains all of the memory management code.
drivers	All of the system's device drivers live in this directory. They are further subdivided into classes of device driver: SPI, block, i2c, pwm, video, usb rtc, tty, gpio, gpu,watchdog etc
fs	All of the file system code. This is further subdivided into directories, one per supported file system, for example ntfs, vfat and ext2.
kernel	The main kernel code. Again, the architecture specific kernel code is in arch/*/kernel.
net	The kernel's networking code.
lib	This directory contains the kernel's library code. The architecture specific library code can be found in arch/*/lib/.
Directory	**Description**

crypto	Device drivers, file system and security all need crypto.
ipc	This directory contains the kernels inter-process communications code.
scripts	This directory contains the scripts (for example awk and tk scripts) that are used when the kernel is configured.

Command lsmod

Show the status of modules in the Linux Kernel. lsmod is a trivial program which nicely formats the contents of the /proc/modules, showing what kernel modules are currently loaded.

"Module" denotes the name of the module.

"Size" denotes the size of the module (not memory used) in Bytes.

"Used by" shows that number of times the module is currently in use by running programs.

```
pi@raspberrypi:~ $ lsmod
Module                  Size  Used by
rfcomm                 49152  4
nls_utf8               16384  1
cifs                  786432  0
cifs_arc4              16384  1 cifs
cifs_md4               16384  1 cifs
cmac                   16384  3
algif_hash             16384  1
aes_arm_bs             24576  2
crypto_simd            16384  1 aes_arm_bs
cryptd                 24576  2 crypto_simd
algif_skcipher         16384  1
af_alg                 28672  6 algif_hash,algif_skcipher
bnep                   20480  2
hci_uart               40960  1
btbcm                  20480  1 hci_uart
bluetooth             409600  31 hci_uart,bnep,btbcm,rfcomm
ecdh_generic           16384  2 bluetooth
ecc                    40960  1 ecdh_generic
8021q                  32768  0
garp                   16384  1 8021q
stp                    16384  1 garp
llc                    16384  2 garp,stp
snd_soc_hdmi_codec     20480  1
spidev                 20480  0
brcmfmac              335872  0
vc4                   290816  5
cec                    49152  1 vc4
brcmutil               20480  1 brcmfmac
drm_kms_helper        274432  2 vc4
sha256_generic         16384  0
cfg80211              765952  1 brcmfmac
snd_soc_core          233472  2 vc4,snd_soc_hdmi_codec
snd_compress           20480  1 snd_soc_core
```

Command modinfo

Show the status of modules in the Linux Kernel. modinfo extracts information from the Linux Kernel modules given on the command line. If the module name is not a file name, then the /lib/modules/$(uname -r)directory is searched by default. Modinfo can understand modules from any of the Linux Kernel architecture.

```
pi@raspberrypi:~ $ modinfo spi_bcm2835
filename:       /lib/modules/5.15.61-v7+/kernel/drivers/spi/spi-bcm2835.ko.xz
license:        GPL
author:         Chris Boot <bootc@bootc.net>
description:    SPI controller driver for Broadcom BCM2835
srcversion:     76B9E825743BD9406356BAA
alias:          of:N*T*Cbrcm,bcm2835-spiC*
alias:          of:N*T*Cbrcm,bcm2835-spi
depends:
intree:         Y
name:           spi_bcm2835
vermagic:       5.15.61-v7+ SMP mod_unload modversions ARMv7 p2v8
parm:           polling_limit_us:time in us to run a transfer in polling mode
 (uint)
pi@raspberrypi:~ $
```

```
pi@raspberrypi:~ $ modinfo bluetooth
filename:       /lib/modules/5.15.61-v7+/kernel/net/bluetooth/bluetooth.ko.xz
alias:          net-pf-31
license:        GPL
version:        2.22
description:    Bluetooth Core ver 2.22
author:         Marcel Holtmann <marcel@holtmann.org>
srcversion:     6EF47A9A5EBBE314CA0F67E
depends:        rfkill,ecdh_generic
intree:         Y
name:           bluetooth
vermagic:       5.15.61-v7+ SMP mod_unload modversions ARMv7 p2v8
parm:           disable_esco:Disable eSCO connection creation (bool)
parm:           disable_ertm:Disable enhanced retransmission mode (bool)
parm:           enable_ecred:Enable enhanced credit flow control mode (bool)
pi@raspberrypi:~ $
```

8.1.4 Linux File System

A Linux file system is a structured collection of files on a disk drive or a partition. A partition is a segment of memory and contains some specific data. In our machine, there can be various partitions of the memory. Generally, every partition contains a file system. The general-purpose computer system needs to store data systematically on hard disks (HDD), SSD, eMMC, SD Card, etc.

Reasons to maintain file system:

Primarily the computer saves data to the RAM storage; it may lose the data if it gets turned off. However, there is non-volatile RAM (Flash RAM and SSD) that is available to maintain the data after the power interruption. Data storage is preferred on hard drives as compared to standard RAM as RAM costs more than disk space. The hard disks costs are dropping gradually comparatively the RAM.

The Linux file system contains the following sections:

The root directory (/). A specific data storage format (EXT3, EXT4, BTRFS, XFS, and so on). A partition or logical volume having a particular file system that can be mounted on a specified mount point on a Linux filesystem.

```
pi@raspberrypi:/ $ ls
bin   dev  home  lost+found  mnt  proc  run   srv  tmp  var
boot  etc  lib   media       opt  root  sbin  sys  usr
pi@raspberrypi:/ $
```

8.1.5 Linux File Hierarchy Structure

/ (root)	**Primary hierarchy root and root directory of the entire file system hierarchy.**
/bin	The /bin directory contains user executable files.
/boot	Contains the static bootloader and kernel executable and configuration files required to boot a Linux computer.
/dev	This directory contains the device files for every hardware device attached to the system. These are not device drivers, rather they are files that represent each device on the computer and facilitate access to those devices.
/etc	Contains the local system configuration files for the host computer.
/home	Home directory storage for user files. Each user has a subdirectory in /home.
/lib	Contains shared library files that are required to boot the system.
/media	A place to mount external removable media devices such as USB thumb drives that may be connected to the host.
/mnt	A temporary mountpoint for regular filesystems (as in not removable media) that can be used while the administrator is repairing or working on a filesystem.
/opt	Optional files such as vendor supplied application programs should be located here.
/root	This is not the root (/) filesystem. It is the home directory for the root user.
/sbin	System binary files. These are executables used for system administration.
/usr	These are shareable, read-only files, including executable binaries and libraries, man files, and other types of documentation.
/var	Variable data files are stored here. This can include things like log files, MySQL, and other database files, web server data files, email inboxes, and much more.

8.1.6 Linux Filesystem Types

There are three major Linux filesystems: ext2, ext3 and ext4.

ext2

- ext2 is suitable for flash drives and USB drives.
- A file can be between 16 GB and 2 TB in size.
- An ext2 filesystem can be between 2 TB and 32 TB in size.
- ext2 filesystems are likely to become corrupt during power failures and computer crashes when data is being saved to the disk.
- ext2 filesystems face data fragmentation issues which hinder performance.

ext3

- ext3 allows journaling. Journaling creates a separate area of the filesystem where all file changes are tracked. The Journal can then be used in case of a power failure or system crash to restore data.
- A directory in ext3 can have up to 32,000 subdirectories.

ext4

- A directory can have up to 64,000 subdirectories.
- A file can be up to 16 TB in size.
- An ext4 filesystem can be up to 1 EB (Exabyte) in size, although most Linux distributions recommend a maximum filesystem size of 100 Tb.
- ext4 reduces fragmentation issues and increases performance.
- It also provides option to turn off the journaling feature.

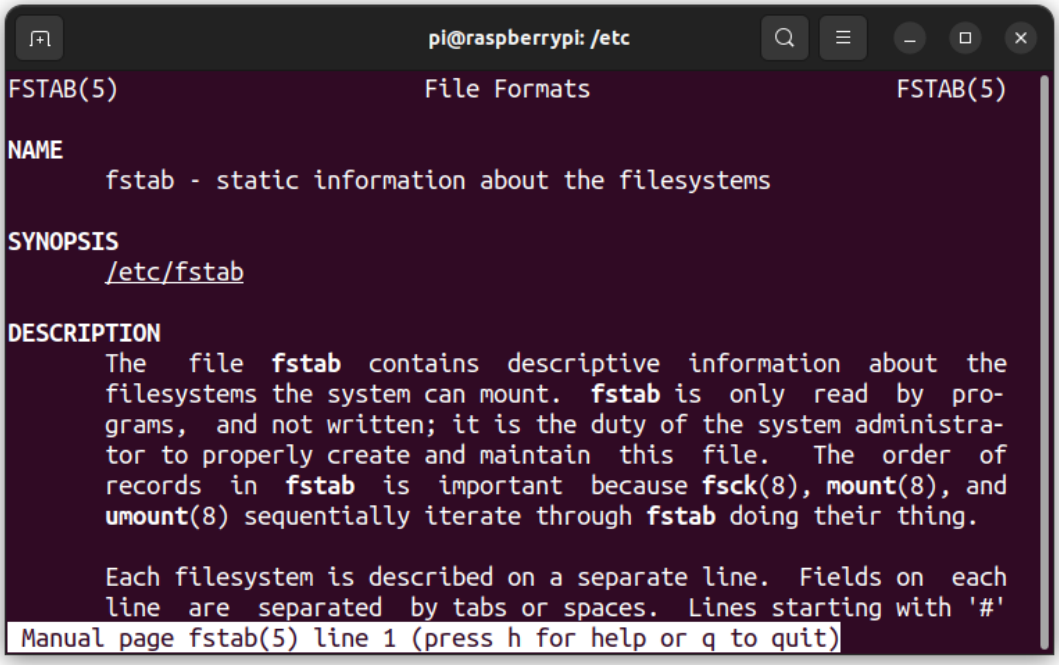
8.1.7 Mounting Filesystem

The filesystem on the disk pack would be logically mounted by the operating system to make the contents available for access by the OS, application programs and users. A mount point is simply a directory. For eg: home filesystem is mounted on the directory /home.

The Linux root filesystem is mounted on the root directory (/) very early in the boot sequence. Other filesystems are mounted later, by the Linux startup programs. Mounting of filesystems during the startup process is managed by the /etc/fstab configuration file. The /etc/fstab file is one of the most important files in a Linux-based system, since it stores static information about filesystems, their mountpoints and mount options. Each line of /etc/fstab contains the necessary settings to mount one partition, drive or network share. The device file, UUID or label or other means of locating the partition or data source. The mount point, where the data is to be attached to the filesystem.

```
pi@raspberrypi:/etc $ cat fstab
proc            /proc           proc    defaults          0       0
PARTUUID=0118764f-01  /boot           vfat    defaults          0       2
PARTUUID=0118764f-02  /               ext4    defaults,noatime  0       1
# a swapfile is not a swap partition, no line here
#   use  dphys-swapfile swap[on|off]  for that
//192.168.0.157 /home/akshay/TrainingMaterial/Module3_SystemProgramming_Using_C cifs guest,_netdev 0 0
pi@raspberrypi:/etc $
```

To read in more details about **/etc/fstab** go through the Linux Manual page.
Use command: **man fstab**



UNIT 8.2: Linux Driver Framework and Filesystem

Unit Objectives

At the end of this unit, you will be able to:

1. Discuss the file system of Linux
2. Elucidate the files/devices invoked in Linux

8.2.1 Linux Driver Framework

In general, any operating system needs a piece of software specific to the device. This piece of software understands the device functionality and is a middle layer between the OS and the Hardware. For eg: Getting input from a keyboard or displaying it on your screen both require devices. Linux provides a mechanism to simplify adding these input and output functions. Linux also provides a large set of tools and utilities to modify and configure how your system and these device drivers interact. Applications and other programs access everything, even hardware, through files. The term used for a special file to access hardware is a "device node". Device nodes facilitate transparent communication between user space applications and computer hardware.

There are two general kinds of device files:

Character special files.

Character devices deliver or accept a stream of characters (bytes) without regard to any other structure.

Some character devices are keyboards and terminals.

Block special files.

A block device is one that stores information in fixed size blocks. Common block sizes are between 128 bytes and 1k bytes. Each block may be read independently of the others, so the device allows random access to each block. Some block devices are hard drives, CD-ROM drives, RAM disks, etc.

Device file location

- All device files are stored in /dev directory.
- Use cd /dev and ls -l command to browse the directory.
- A character device is marked with a c as the first letter of the permissions strings.
- A block device is marked with a b as the first letter of the permissions strings.

```
pi@raspberrypi:/dev $ ls -l
total 0
crw-r--r--  1 root root     10, 235 Dec  3 14:17 autofs
drwxr-xr-x  2 root root         580 Dec  3 14:17 block
crw-------  1 root root     10, 234 Dec  3 14:17 btrfs-control
drwxr-xr-x  3 root root          60 Jan  1  1970 bus
crw-------  1 root root     10, 127 Dec  3 14:17 cachefiles
crw-rw----  1 root video    239,   0 Dec  3 14:36 cec0
drwxr-xr-x  2 root root        3080 Dec  3 14:37 char
crw--w----  1 root tty        5,   1 Dec  3 14:17 console
crw-------  1 root root     10, 203 Dec  3 14:17 cuse
drwxr-xr-x  7 root root         140 Dec  3 14:17 disk
drwxr-xr-x  2 root root          80 Jan  1  1970 dma_heap
drwxr-xr-x  3 root root         100 Dec  3 14:36 dri
lrwxrwxrwx  1 root root          13 Sep 13 07:28 fd -> /proc/self/fd
crw-rw-rw-  1 root root      1,   7 Dec  3 14:17 full
crw-rw-rw-  1 root root     10, 229 Dec  3 14:17 fuse
crw-rw----  1 root gpio    254,   0 Dec  3 14:17 gpiochip0
crw-rw----  1 root gpio    254,   1 Dec  3 14:17 gpiochip1
crw-rw----  1 root gpio    254,   2 Dec  3 14:17 gpiochip2
crw-rw----  1 root gpio    245,   0 Dec  3 14:17 gpiomem
crw-------  1 root root     10, 183 Dec  3 14:17 hwrng
crw-rw----  1 root i2c      89,   1 Dec  3 14:36 i2c-1
crw-rw----  1 root i2c      89,   2 Dec  3 14:36 i2c-2
lrwxrwxrwx  1 root root          12 Sep 13 07:28 initctl -> /run/initctl
drwxr-xr-x  3 root root         100 Dec  3 14:36 input
crw-r--r--  1 root root      1,  11 Dec  3 14:17 kmsg
lrwxrwxrwx  1 root root          28 Sep 13 07:28 log -> /run/systemd/journal/dev-log
brw-rw----  1 root disk      7,   0 Dec  3 14:17 loop0
brw-rw----  1 root disk      7,   1 Dec  3 14:17 loop1
brw-rw----  1 root disk      7,   2 Dec  3 14:17 loop2
brw-rw----  1 root disk      7,   3 Dec  3 14:17 loop3
brw-rw----  1 root disk      7,   4 Dec  3 14:17 loop4
brw-rw----  1 root disk      7,   5 Dec  3 14:17 loop5
brw-rw----  1 root disk      7,   6 Dec  3 14:17 loop6
brw-rw----  1 root disk      7,   7 Dec  3 14:17 loop7
```

8.2.2 Introduction To chardev GPIO and Libgpiod on the Raspberry PI

chardev GPIO:

- We will access GPIO is via the “descriptor-based” character device.
- The interface is exposed at /dev/gpiochipN, where N is the chip number.

Libgpiod:

- Libgpiod (Library General Purpose Input/Output device) provides both API calls for use in your own programs. gpiod is a set of tools for interacting with the Linux GPIO character device that uses libgpiod library.

Install the tools with: sudo apt-get install gpiod libgpiod-dev libgpiod-do

```
pi@raspberrypi:~ $ ls /dev/gpiochip*
/dev/gpiochip0  /dev/gpiochip1  /dev/gpiochip2
pi@raspberrypi:~ $
```

Following example usage is on a Raspberry PI 3 Model B V1.2.
The following are six user-mode applications to manipulate GPIO:

gpiodetect:

Detect/list GPIO character devices:

gpioinfo:

List all lines of specified gpiochips, their names, consumers, direction, active state and additional flags.

gpioget:

Read values of specified GPIO lines

gpioset:

Set values of specified GPIO lines, potentially keep the lines exported and wait until timeout, user input or signal

gpiofind:

Find the gpiochip name and line offset given the line name

gpiomon:

Wait for events on GPIO lines, specify which events to watch, how many events to process before exiting or if the events should be reported to the console

```
pi@raspberrypi:~ $ gpiodetect
gpiochip0 [pinctrl-bcm2835] (54 lines)
gpiochip1 [brcmvirt-gpio] (2 lines)
gpiochip2 [raspberrypi-exp-gpio] (8 lines)
pi@raspberrypi:~ $
```

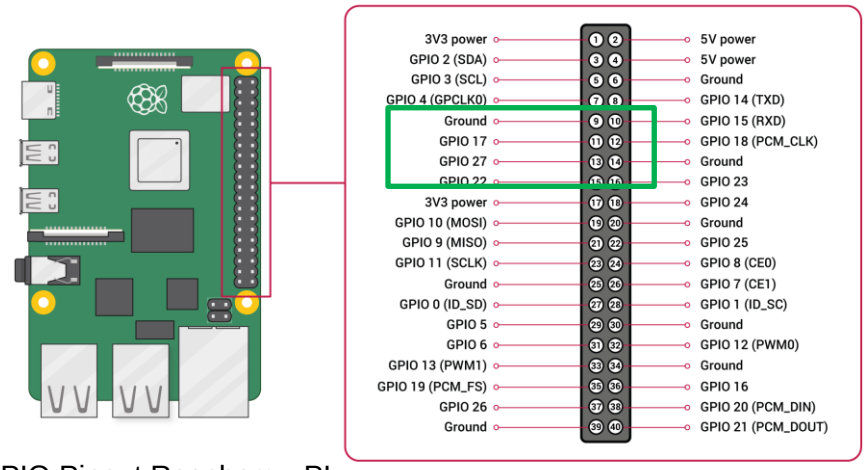
```
pi@raspberrypi:~ $ gpioinfo pinctrl-bcm2835
gpiochip0 - 54 lines:
        line   0:      "ID_SDA"       unused   input  active-high
        line   1:      "ID_SCL"       unused   input  active-high
        line   2:        "SDA1"       unused   input  active-high
        line   3:        "SCL1"       unused   input  active-high
        line   4:  "GPIO_GCLK"  "onewire@0"  output  active-high [used open-drain]
        line   5:       "GPIO5"       unused   input  active-high
        line   6:       "GPIO6"       unused   input  active-high
        line   7:  "SPI_CE1_N"   "spi0 CS1"  output   active-low [used]
        line   8:  "SPI_CE0_N"   "spi0 CS0"  output   active-low [used]
        line   9:   "SPI_MISO"       unused   input  active-high
        line  10:   "SPI_MOSI"       unused   input  active-high
        line  11:   "SPI_SCLK"       unused   input  active-high
        line  12:      "GPIO12"       unused   input  active-high
        line  13:      "GPIO13"       unused   input  active-high
        line  14:        "TXD1"       unused   input  active-high
        line  15:        "RXD1"       unused   input  active-high
        line  16:      "GPIO16"       unused   input  active-high
        line  17:      "GPIO17"       unused   input  active-high
```

gpioget

- Consider we want to read input status of GPIO 17.
- GPIO 17 is comes under gpiochip0.
- Attach a Switch to GPIO 17 pin.
- Run the command.

gpioget gpiochip0 17

- Try changing the switch state and run the command again
- See the result below:



GPIO Pinout Raspberry -PI

Fig 8.2.2 gpioget gpiochip 017

```
pi@raspberrypi:~ $ gpioget gpiochip0 17
0
pi@raspberrypi:~ $ gpioget gpiochip0 17
1
pi@raspberrypi:~ $
```

gpiomon

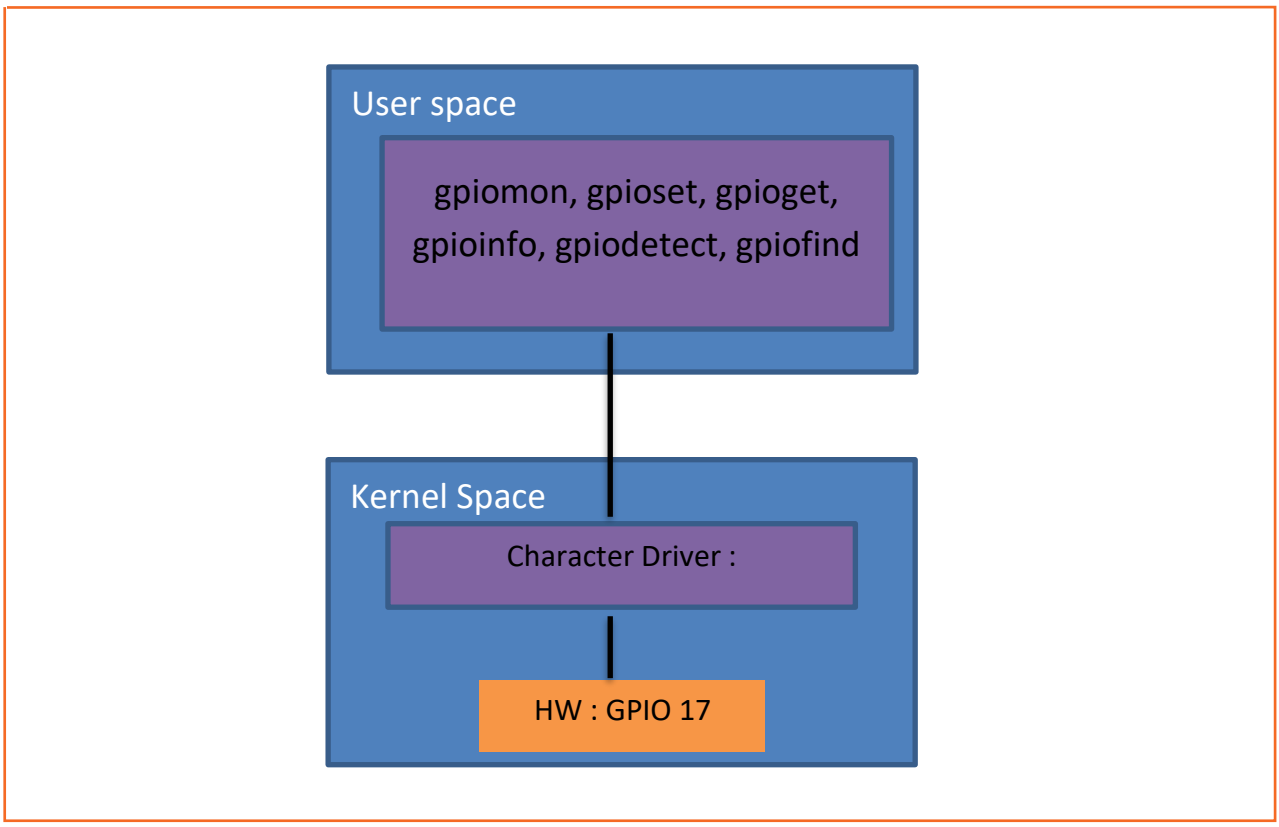
- Test wait for three rising edge events on a single GPIO 17 line:
- Run the following command:

gpiomon --num-events=3 --rising-edge gpiochip0 17

How these tools are working?

- All the above tools run from user application.
- Character device file: /dev/gpiochip0 has allows access to the GPIO17.

```
pi@raspberrypi:~ $ gpiomon --num-events=3 --rising-edge gpiochip0 17
event:  RISING EDGE offset: 17 timestamp: [   24989.157302663]
event:  RISING EDGE offset: 17 timestamp: [   24989.269821761]
event:  RISING EDGE offset: 17 timestamp: [   24989.961357286]
pi@raspberrypi:~ $
```


User space
gpiomon, gpioset, gpioget,
gpioinfo, gpiodetect, gpiofind
Kernel Space
Character Driver :
HW : GPIO 17

UNIT 8.3: Networking Sockets

Unit Objectives

At the end of this unit, you will be able to:

1. Develop a program using socket to establish the communication between two systems.

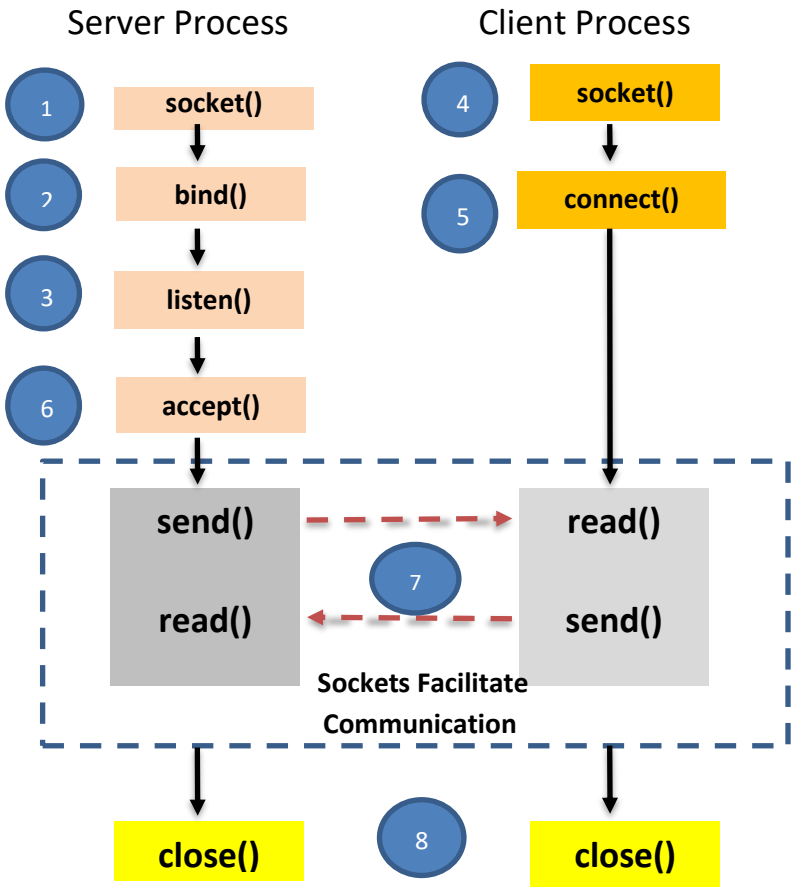
8.3.1 Networking Sockets

In user space, the abstraction of network communication is the socket. Sockets are a way to enable inter-process communication between programs running on a server, or between programs running on separate servers. Communication between servers relies on network sockets, which use the Internet Protocol (IP) to encapsulate and handle sending and receiving data. Network sockets on both clients and servers are referred to by their socket address. An address is a unique combination of a transport protocol like the Transmission Control Protocol (TCP) or User Datagram Protocol (UDP), an IP address, and a port number.

Different types of sockets that are used for inter-process communication:

- Stream sockets, which use TCP as their underlying transport protocol
- Datagram sockets, which use UDP as their underlying transport protocol
- Unix Domain Sockets, which use local files to send and receive data instead of network interfaces and IP packets.

Function Call	Description
socket()	To create a socket
bind()	Used to associate the socket with local address i.e. IP Address, port and address family.
listen()	Ready to receive a connection
connect()	Used by the client application to establish a connection to a server.
accept()	Confirmation, it is like accepting to receive a call from a sender
send()	To send data
read()	To receive data
close()	To close a connection



Stream sockets are connection oriented, which means that packets sent to and received from a network socket are delivered by the host operating system in order for processing by an application. Network based stream sockets typically use the Transmission Control Protocol (TCP) to encapsulate and transmit data over a network interface. TCP is designed to be a reliable network protocol that relies on a stateful connection.

Example Program:

test_tcp_server.c -:

Listens for socket connection on **localhost IP Address: 127.0.0.1** on **Port Number: 3456.**

Replies to the client once message is received from client over socket connection.

test_tcp_client.c -:

Connects to Server over **localhost IP Address: 127.0.0.1** on **Port Number: 3456.**

Sends a message to server on connection.


TCP/IP Connection:
127.0.0.1:3456
akshay@akshayv:Chapter8_Linux_Fundamentals$ gcc test_tcp_server.c -o tcp_server
akshay@akshayv:Chapter8_Linux_Fundamentals$ gcc test_tcp_client.c -o tcp_client
akshay@akshayv:Chapter8_Linux_Fundamentals$ ls
tcp_client tcp_server test_tcp_client.c test_tcp_server.c
akshay@akshayv:Chapter8_Linux_Fundamentals$ ./tcp_server
Server Application Started!!!!!
Hi I am client!!
Hello message sent to Client
akshay@akshayv:Chapter8_Linux_Fundamentals$
akshay@akshayv:Chapter8_Linux_Fundamentals$ ls
tcp_client tcp_server test_tcp_client.c test_tcp_server.c
akshay@akshayv:Chapter8_Linux_Fundamentals$ ./tcp_client
Client Application Started!!!!!
Hello message sent
Hi I am server!!!
akshay@akshayv:Chapter8_Linux_Fundamentals$

UNIT 8.4: Debug Tools

Unit Objectives

At the end of this unit, you will be able to:

1. Enter into debug mode
2. Use the debug commands to analyze the codes
3. Explain the kernel debug mode

8.4.1 Debug Tools

GDB is a debugger for **C** and **C++**. GDB offers extensive facilities for **tracing** and **altering** the execution of computer programs. The user can monitor and modify the values of programs' internal variables, and even call functions independently of the program's normal behavior.

GDB allows you to do things like:

- **Add breakpoints** - Run the program up to a certain point then stop
- Print out the values of certain variables at that point.
- Step through the program one line at a time and print out the values of each variable after executing each lin**e.**

GDB is by default typically compiled to target the same architecture as the host system.

- The **host** architecture: where the GDB program itself is run. The **target** architecture: where the program being debugged is run

GDB Operation:

Compile with the "-g" option when using GCC to compile the code. It generates added information in the object code, so the debugger can match a line of source code with the step of execution. Do not use compiler optimization directive such as "-O" or "-O2" which rearrange computing operations to gain speed as this reordering will not match the order of execution in the source code and it may be impossible to follow. control+c: Stop execution. It can stop program anywhere, in your source or a C library or anywhere. To start GDB session use command: gdb. /<Program Name to debug>

GDB command completion: Use TAB key

- Press TAB twice to see all available options if more than one option is available or type

GDB Commands:

Start and Stop

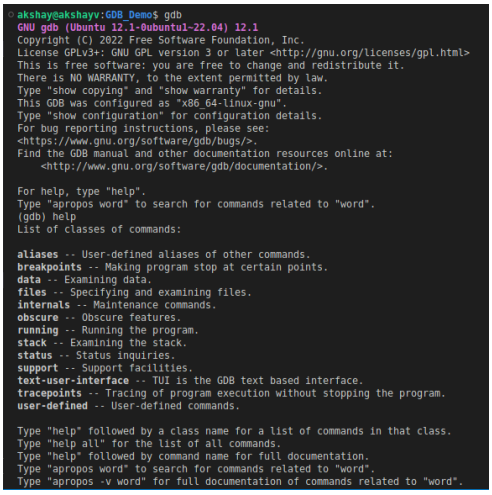
Start and stop	Description
run r run command-line-arguments run < infile > outfile	Start program execution from the beginning of the program. The command **break main** will get you started. Also allows basic I/O redirection.
continue c c [ignore-count]	Continue execution to next break point. [ignore-count] - Igonre number of counts breakpoint hits
kill	Stop program execution.
quit q	Exit GDB debugger.

Help and Info Commands:

Will provide some help and info regarding the commands and usage:

Command	Description
help	List gdb command topics.
help topic-classes	List gdb command within class.
help command	Command description. eg help show to list the show commands
apropos search-word	Search for commands and command topics containing search-word.
info args i args	List program command line arguments
info breakpoints	List breakpoints
info break	List breakpoint numbers.
info break breakpoint-number	List info about specific breakpoint.
info watchpoints	List breakpoints
info registers	List registers in use
info threads	List threads in use

Command	Description
info set	List set-able option



GDB Commands:

Breakpoint and Watch:

Break and Watch	Description
break function-name break line-number break Class Name: function Name	Suspend program at specified function of line number.
break filename: function	Don't specify path, just the file name and function name.
break filename: line-number	Don't specify path, just the file name and line number. break Directory/Path/filename.cpp:62
break *address	Suspend processing at an instruction address. Used when you do not have source.
break line-number if condition	Where condition is an expression. i.e., x > 10 Suspend when Boolean expression is true.
break line thread thread-number	Break in thread at specified line number. Use info threads to display thread numbers.
tbreak	Temporary break. Break once only. Break is then removed. See "break" above for options.
watch condition	Suspend processing when condition is met. i.e., x > 100
clear clear function clear line-number	Delete breakpoints as identified by command option. Delete all breakpoints in function Delete breakpoints at a given line
delete d	Delete all breakpoints, watchpoints, or catchpoints.

disable breakpoint-number-or-range enable breakpoint-number-or-range	Does not delete breakpoints. Just enables/disables them. Example: Show breakpoints: info break Disable: disable 2-8
enable breakpoint-number once	Enables once
continue c	Continue executing until next break point/watchpoint.
continue number	Continue but ignore current breakpoint number times. Useful for breakpoints within a loop.
finish	Continue to end of function.

GDB Commands:

Stack:

Stack	Description
backtrace bt bt inner-function-nesting-depth bt -outer-function-nesting-depth	Show trace of where you are currently. Which functions you are in. Prints stack backtrace.
backtrace full	Print values of local variables.
frame frame number f number	Show current stack frame (function where you are stopped) Select frame number. (Can also user up/down to navigate frames)
up down up number down number	Move up a single frame (element in the call stack) Move down a single frame Move up/down the specified number of frames in the stack.
info frame	List address, language, address of arguments/local variables and which registers were saved in frame.
info args info locals info catch	Info arguments of selected frame, local variables and exception handlers.

GDB Commands:

Line Execution:

Use to execute code line by line.

Step into/over a function.

Run program till line number.

Line Execution	Description
step s step number-of-steps-to-perform	Step to next line of code. Will step into a function.
next n next number	Execute next line of code. Will not enter functions.
until until line-number	Continue processing until you reach a specified line number. Also: function name, address, filename:function or filename:line-number.
info signals info handle handle SIGNAL-NAME option	Perform the following option when signal recieved: nostop, stop, print, noprint, pass/noignore or nopass/ignore
where	Shows current line number and which function you are in.

Source Code Commands:

To print lines from a source file during the debug session

Source Code	Description
list l list line-number list function list - list start#,end# list filename:function	List source code. To print lines from a source file.
set listsize count show listsize	Number of lines listed when list command given.

Consider sample program: demo_sampleCode.c

The program: Prints n-th Fibonacci Number

Compile the code with command:

gcc demo_sampleCode.c -g -o demo_sampleCode

Start a GDB Session with comand:

gdb --args ./demo_sampleCode 9

```
akshay@akshayv:GDB_Demo$ gcc demo_sampleCode.c -g -o demo_sampleCode
akshay@akshayv:GDB_Demo$ ./demo_sampleCode  9
Print 9-th Fibonacci Number:
Output : 34
akshay@akshayv:GDB_Demo$
```

This is where we type commands:

```
akshay@akshayv:GDB_Demo$ gdb ./demo_sampleCode
GNU gdb (Ubuntu 12.1-0ubuntu1~22.04) 12.1
Copyright (C) 2022 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
Type "show copying" and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<https://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
    <http://www.gnu.org/software/gdb/documentation/>.

For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from ./demo_sampleCode...
(gdb)
(gdb)
(gdb) list fib
2       #include <stdlib.h>
3       #include <string.h>
4
5       // Find Fibonacci with Recursion
6       int fib(int n)
7       {
8           if (n <= 1)
9               return n;
10          return fib(n - 1) + fib(n - 2);
11      }
(gdb)
```

Source Code Command:

To view the source code, we can use list command:

To view fib () function use command: list

```
Reading symbols from ./demo_sampleCode...
(gdb)
(gdb) break 15
Breakpoint 1 at 0x11db: file demo_sampleCode.c, line 15.
(gdb) run
Starting program: /home/akshay/TrainingMaterial/Module3_SystemProgramming_Using_C/Chapter2_Building_an_Executable/GDB_Demo/demo_sampleCode 9
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".

Breakpoint 1, main (argc=2, argv=0x7fffffffdbf8) at demo_sampleCode.c:15
15          if(argc < 2){
(gdb) info args
argc = 2
argv = 0x7fffffffdbf8
(gdb) print *argv@argc
$1 = {
  0x7fffffffdf93 "/home/akshay/TrainingMaterial/Module3_SystemProgramming_Using_C/Chapter2_Building_an_Executable/GDB_Demo/demo_sampleCode", 0x7fffffffe00c "9"}
(gdb) next
20          printf("Print %d-th Fibonacci Number:\r\n", atoi(argv[1]));
(gdb) n
Print 9-th Fibonacci Number:
21          printf("Output : %d\r\n", fib(atoi(argv[1])));
(gdb) break 8
Breakpoint 2 at 0x555555555199: file demo_sampleCode.c, line 8.
(gdb) continue
Continuing.

Breakpoint 2, fib (n=9) at demo_sampleCode.c:8
8           if (n <= 1)
(gdb) print n
$2 = 9
(gdb) c
Continuing.

Breakpoint 2, fib (n=8) at demo_sampleCode.c:8
8           if (n <= 1)
(gdb) c
Continuing.

Breakpoint 2, fib (n=7) at demo_sampleCode.c:8
8           if (n <= 1)
(gdb) p n
$3 = 7
(gdb) c 3
Will ignore next 2 crossings of breakpoint 2.  Continuing.

Breakpoint 2, fib (n=4) at demo_sampleCode.c:8
8           if (n <= 1)
(gdb) p n
$4 = 4
```

Break Point at line 15

Run – start program

Code reach Breakpoint 1

info args : to get details about arguents to program

print *argv@argc : print arguments

1

2

3

4

5

1. "**next**" or "**n**" **:** Execute next line
2. **break 8** : add breakpoint at line 8
3. "**continue**" or "**c**" : Execute until next breakpoint
4. "**print n**" or "**p n**" : Print value at variable **n**
5. **c 3** : ignore break point 3 times

bt: Show trace of where you are currently. Which functions you are in.

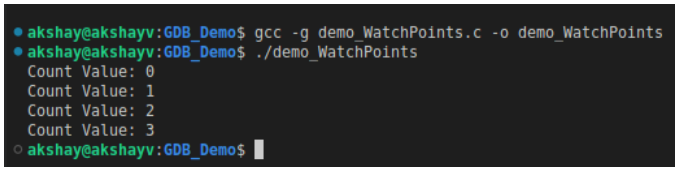
where: Print out the call stack including files and line numbers.

Helpful to see the function calling sequence of

GDB – Traverse Call Stack

1. **up:** Go up the stack i.e., go to the line that called the function you are currently in.
2. **up**
3. **p n**: print value on n (After 2 **up** commands code is currently in **fib(n = 6)** function)
4. **down 2:** Go down the **stack 2 frames**
5. **p n**: print value on n (After 1 **up** command and down 2 frames code is currently in **fib(n =5)** function)

Consider the following source code: demo_WatchPoints.c



Able to set a watchpoint on a variable in order to break a program when a variable changes.
Use display to automatically print how variables change throughout the program's execution.

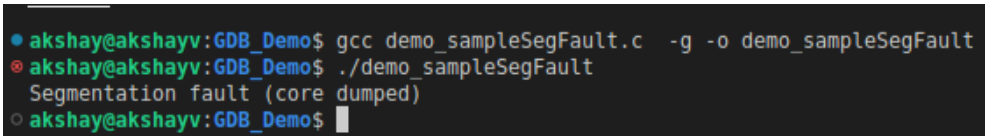
```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main( int argc, char *argv[] )
{
  int count = 0;
  for(int i = 0; i < 4; i++)
  {
    printf("Count Value: %d\r\n", count);
    count += 1;
  }
  return 0;
}
```

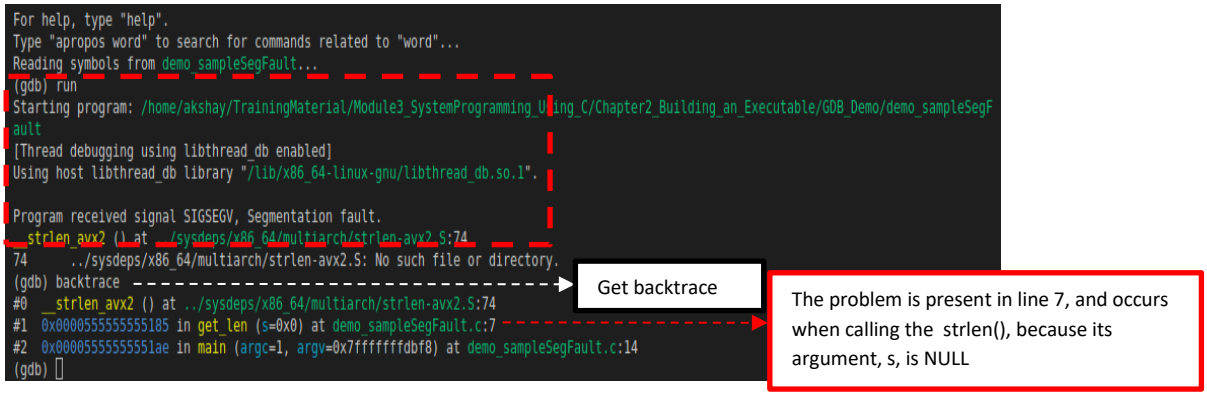
Consider the following source code: demo_sampleSegFault.c

Compile the code with below:

gcc demo_sampleSegFault.c -g -o demo_sampleSegFault

-g flag in order to include appropriate debug information on the binary generated, thus making it possible to inspect it using GDB. When we execute the program, generates a segmentation fault GDB can be used to inspect the problem.





```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

size_t get_len( const char *s )
{
  return strlen( s );
}

int main( int argc, char *argv[] )
{
  const char *a = NULL;

  printf( "size of a = %lu\r\n", get_len(a) );

  return 0;
}
```

8.4.2 Debug Tools: KDB

The kernel has two debuggers: KDB and KGDB. The KDB is kernel debugger is integrated into the kernel and allows full control of the system while a debugging session is in progress. The KGDB is kernel GDB server. It allows a second computer to run GDB and debug the kernel. KDB is simple shell-style interface, which you can use on a system with a keyboard or serial console and Host machine.

We can use KDB to inspect:

- Memory registers
- Process lists
- Dmesg - prints the message buffer of the kernel.
- Set breakpoints to stop in a certain location or device drivers.
- Backtrace any process
- List modules

KDB is not designed as a source level debugger. KDB is mainly focused at doing some analysis to help in development or diagnosing kernel problems. KGDB is used as a source level debugger for the Linux kernel.

Serial Port Enable:

We need to use USB to Serial convertor to UART Pins on Raspberry Pi to connect serially from Host machine. Enable the serial port on Raspberry Pi through configuration. From host machine we can use Screen or Minicom to the connect to Pi.

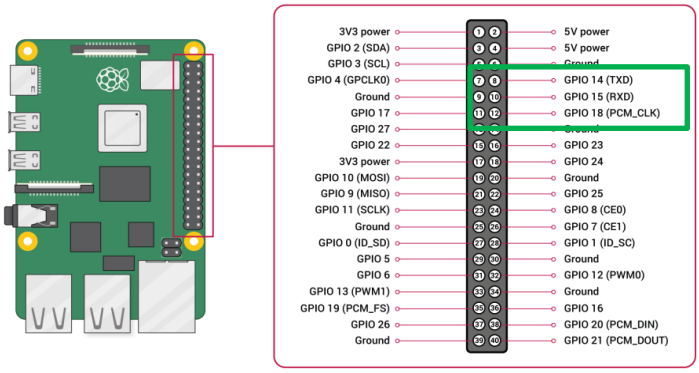
Use command:

screen /dev/ttyXXX 115200

ttyXXX is serial device assigned to USB to Serial Convertor

We will see the Login prompt, login using the correct credentials.

You can see the expected output.



GPIO Pinout Raspberry -PI

SSH Login Enable

- Remote access will be required.
- It makes easy to transfer files to Raspbery PI.
- Follow the document for to enable remote access.

```
pi login: pi
Password:
Linux pi 5.15.80-v7+ #1602 SMP Tue Nov 29 14:42:58 GMT 2022 armv7l

The programs included with the Debian GNU/Linux system are free software;
the exact distribution terms for each program are described in the
individual files in /usr/share/doc/*/copyright.

Debian GNU/Linux comes with ABSOLUTELY NO WARRANTY, to the extent
permitted by applicable law.
Last login: Wed Dec 14 20:35:35 IST 2022 on tty1
pi@pi:~$
```

1

Kernel config options for kdb

To use KDB we must implement shell and add some helper functions into kernel.

We need to recompile the kernel by enabling the following symbols before building the kernel:

CONFIG_FRAME_POINTER=y

CONFIG_KGDB=y

CONFIG_KGDB_SERIAL_CONSOLE=y

CONFIG_KGDB_KDB=y

CONFIG_KDB_KEYBOARD=y

To build kernel follow the Rapsberry Pi Documentation: The Linux Kernel

kdb/kgdboc arguments

Using loadable module or built-in

For kernel built-in: Use the kernel boot argument: kgdboc=<tty-device>,[baud] in /boot/cmdline.txt file. For kernel loadable module: Use the command: modprobe kgdboc kgdboc=<tty-device>,[baud]

modprobe kgdboc kgdboc=ttyS0,115200

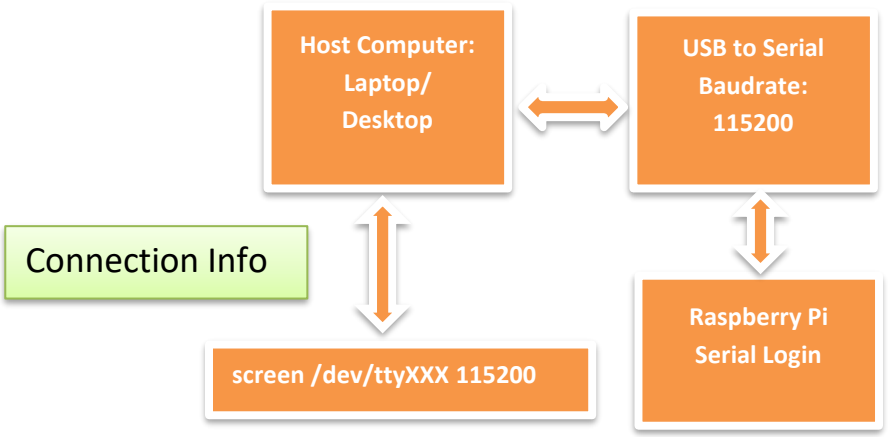
Using KDB

Ubuntu based Host machine used for demo

Connect using serial to Raspberry Pi as shown >>>

Root Access:

- Login as root.
- Or use a super user session with command: sudo




```
root@pi:/home/pi# echo ttyS0 > /sys/module/kgdboc/parameters/kgdboc
root@pi:/home/pi# echo g > /proc/sysrq-trigger
[  209.032572] sysrq: DEBUG

Entering kdb (current=0xb0815e80, pid 978) on processor 0 due to Keyboard Entry
[0]kdb>

[0]kdb>
```

Screen Terminal Access

Configure kgdboc at runtime with sysfs

- At run time enable or disable kgdboc by echoing a parameter into the sysfs.
- Enable kgdboc on ttyS0:

echo ttyS0 > /sys/module/kgdboc/parameters/kgdboc

- Disable kgdboc:

echo "" > /sys/module/kgdboc/parameters/kgdboc

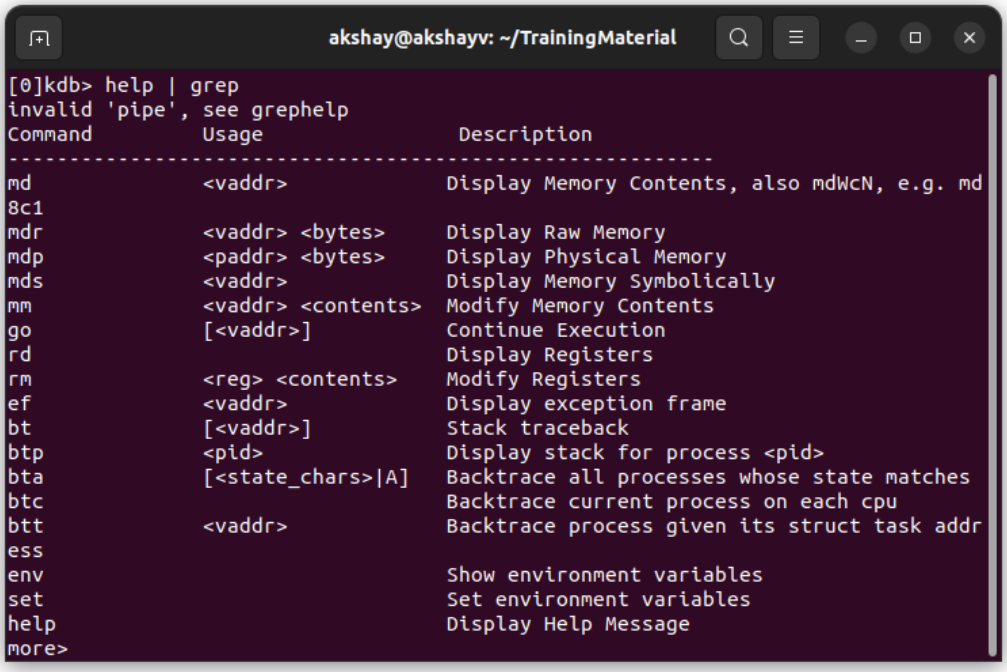
Enter the KDB:

- We can enter the KDB manually or by waiting for some OOPS/ fault in kernel.
- Manually entering kdb : echo g > /proc/sysrq-trigger

KDB Commands:

Command	Description
lsmod	To show all the loaded modules
ps	To displays the active processes
ps A	To show all the processes
summary	To show kernel version info and memory usage
bt	To get a backtrace of the current process using dump_stack ()
dmesg	To view the kernel syslog buffer
go	To continue the system
help	To read the help for different commands
reboot	To reboot device

KDB > help:



KDB > lsmod:

This command will show all the loaded modules into kernel

```
[0]kdb> lsmod
Module                  Size  modstruct     Used by
nls_utf8               16384  0x7f673040    1  (Live) 0x7f671000 [ ]
cifs                  786432  0x7f61eb80    0  (Live) 0x7f586000 [ ]
cifs_arc4              16384  0x7f583000    1  (Live) 0x7f581000 [ cifs_arc4 ]
cifs_md4               16384  0x7f57e000    1  (Live) 0x7f57c000 [ cifs_md4 ]
cmac                   16384  0x7f56d0c0    0  (Live) 0x7f56b000 [ ]
algif_hash             16384  0x7f53c100    0  (Live) 0x7f53a000 [ ]
aes_arm_bs             24576  0x7f579900    0  (Live) 0x7f575000 [ ]
crypto_simd            16384  0x7f572000    1  (Live) 0x7f570000 [ crypto_simd ]
cryptd                 24576  0x7f5670c0    1  (Live) 0x7f564000 [ cryptd ]
algif_skcipher         16384  0x7f541100    0  (Live) 0x7f53f000 [ ]
af_alg                 28672  0x7f536140    2  (Live) 0x7f532000 [ af_alg af_alg ]
bnep                   20480  0x7f52f180    2  (Live) 0x7f52c000 [ ]
hci_uart               40960  0x7f560180    1  (Live) 0x7f559000 [ ]
btbcm                  20480  0x7f556000    1  (Live) 0x7f553000 [ btbcm ]
bluetooth             409600  0x7f518440    9  (Live) 0x7f4c7000 [ bluetooth bluetooth bluetooth ]
ecdh_generic           16384  0x7f4c4540    1  (Live) 0x7f4c2000 [ ecdh_generic ]
ecc                    40960  0x7f4be380    1  (Live) 0x7f4b7000 [ ecc ]
8021q                  32768  0x7f353100    0  (Live) 0x7f34e000 [ ]
garp                   16384  0x7f34b040    1  (Live) 0x7f349000 [ garp ]
stp                    16384  0x7f346080    1  (Live) 0x7f344000 [ stp ]
llc                    16384  0x7f341080    2  (Live) 0x7f33f000 [ llc llc ]
more>
```

KDB > ps:

This command will show all active task list

KDB > summary:

To show:

- Kernel version info
- Memory usage.

```
[0]kdb> summary
sysname    Linux
release    5.15.80-v7+
version    #1602 SMP Tue Nov 29 14:42:58 GMT 2022
machine    armv7l
nodename   pi
domainname (none)
date       2022-12-14 15:50:24 tz_minuteswest 0
uptime     00:19
load avg   1.39 0.95 0.41

MemTotal:         944268 kB
MemFree:          348592 kB
Buffers:           29492 kB
[0]kdb>
```

```
[2]kdb> ps
3 idle processes (state -) and
102 sleeping system daemon (state [ims]) processes suppressed,
use 'ps A' to see all.
Task Addr       Pid   Parent [*] cpu State Thread     Command
0x8e636e40     1339     1337  1    2   R  0x8e637d44 *bash

0x81548000        1        0  0    2   S  0x81548f04  systemd
0x82bb6e40      127        1  0    1   S  0x82bb7d44  systemd-journal
0x8368bf00      158        1  0    3   S  0x8368ce04  systemd-udevd
0x8503ee40      315        1  0    3   S  0x8503fd44  systemd-timesyn
0x86b00fc0      366        1  0    3   S  0x86b01ec4  sd-resolve
0x86b00000      394        1  0    0   S  0x86b00f04  avahi-daemon
0x86b03f00      396        1  0    2   S  0x86b04e04  dbus-daemon
0x82bb4ec0      417      394  0    3   S  0x82bb5dc4  avahi-daemon
0x82bb2f40      418        1  0    0   S  0x82bb3e44  polkitd
0x8356ee40      457        1  0    2   S  0x8356fd44  gmain
0x83c82f40      506        1  0    3   S  0x83c83e44  gdbus
0x817e8000      437        1  0    2   S  0x817e8f04  rsyslogd
0x83688000      481        1  0    1   S  0x83688f04  in:imuxsock
0x83c86e40      482        1  0    3   S  0x83c87d44  in:imklog
0x82bb1f80      483        1  0    3   S  0x82bb2e84  rs:main Q:Reg
0x87659f80      455        1  0    3   S  0x8765ae84  systemd-logind
more>
```

KDB > dmesg:

To view the kernel syslog buffer. It prints the message buffer of the kernel. The output shows all messages produced by the device drivers. Kernel has own printf kernel API called printk, all those messages can be viewed in the dmesg output

```
                                Dump ftrace log; -skip dumps last #entries
[0]kdb>    dmesg
<6>[    0.000000] Booting Linux on physical CPU 0x0

<6>[    0.000000] CPU: ARMv7 Processor [410fd034] revision 4 (ARMv7), cr=10c5383d
<6>[    0.000000] CPU: div instructions available: patching division code
<6>[    0.000000] CPU: PIPT / VIPT nonaliasing data cache, VIPT aliasing instruction cache
<6>[    0.000000] OF: fdt: Machine model: Raspberry Pi 3 Model B Rev 1.2
<5>[    0.000000] random: crng init done
<6>[    0.000000] Memory policy: Data cache writealloc
<6>[    0.000000] Reserved memory: created CMA memory pool at 0x1ec00000, size 256 MiB
<6>[    0.000000] OF: reserved mem: initialized node linux,cma, compatible id shared-dma-pool
<6>[    0.000000] Zone ranges:
<6>[    0.000000]   DMA      [mem 0x0000000000000000-0x00000003b3fffff]
<6>[    0.000000]   Normal   empty
<6>[    0.000000] Movable zone start for each node
<6>[    0.000000] Early memory node ranges
<6>[    0.000000]   node   0: [mem 0x0000000000000000-0x00000003b3fffff]
<6>[    0.000000] Initmem setup node 0 [mem 0x0000000000000000-0x00000003b3fffff]
more>
```

KDB > demo:

For demo we will follow the steps:

- Study the IOCTL System call
- Write Simple Device Driver that uses IOCTL system call
- Cross Compile setup for driver
- How to load / unload device driver
- Use KDB to put breakpoint in the device driver

Expected learning:

- How to simple write device driver.
- Cross compile tools
- How to use KDB to debug into kernel space

8.4.2 System Calls: ioctl ()

Ioctl () is an abbreviation of input/output control. It is a system call for device-specific input/output operations and other operations which cannot be expressed by regular system calls. It takes a parameter specifying a request code; the effect of a call depends completely on the request code. Request codes are often device-specific. For instance, a CD-ROM device driver which can instruct a physical device to eject a disc would provide an ioctl request code to do so.

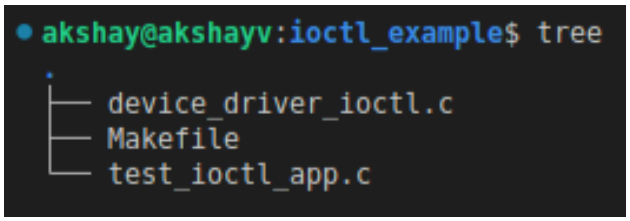
Steps involved in IOCTL:

- Create IOCTL command in the driver
- Write IOCTL function in the driver
- Create IOCTL command in a Userspace application
- Use the IOCTL system call in a Userspace

Example Program:

device_driver_ioctl.c: Simple Device driver with IOCTL

test_ioctl_app.c: Test app which uses IOCTL request.



Define the ioctl command:

#define “Name of IOCTL” _IOx(num1, num2, argument type)

where IOx can be:

“IO“: an ioctl with no parameters

“IOW“: an ioctl with write parameters (copy_from_user)

“IOR“: an ioctl with read parameters (copy_to_user)

“IOWR“: an ioctl with both write and read parameters

num1: is a unique number or character that will differentiate our set of ioctl calls from the other ioctl calls. sometimes the major number for the device is used here.
num2: is the number that is assigned to the ioctl. This is used to differentiate the commands from one another.
argument type: Type of data.
Command example used in our driver:

```
#define WR_VALUE _IOW('a','a',int32_t*)
#define RD_VALUE _IOR('a','b',int32_t*)
```

Write IOCTL function in the driver:
We need to add the ioctl function to our driver. The prototype is as shown in image:
"file": the file pointer to the file that was passed by the application.
"cmd": the ioctl command that was called from the userspace.
"arg": are the arguments passed from the userspace.

- Within function we implement all the commands we defined.
- To inform kernel that ioctl call is implemented in test_ioctl ():

We make fops poiter unlocked_ioctl point to test_ioctl.

```
static long test_ioctl(struct file *file, unsigned int cmd, unsigned long arg)
{
        switch(cmd) {
                case WR_VALUE:
                        if( copy_from_user(&value ,(int32_t*) arg, sizeof(value)) )
                        {
                                pr_err("Data Write : Err!\n");
                        }
                        pr_info("Driver Write Value = %d\n", value);
                        break;
                case RD_VALUE:
                        if( copy_to_user((int32_t*) arg, &value, sizeof(value)) )
                        {
                                pr_err("Data Read : Err!\n");
                        }
                        pr_info("Driver Value to user = %d\n", value);
                        break;
                default:
                        pr_info("Default\n");
                        break;
        }
        return 0;
}
```

```
/*
** File operation sturcture
*/
static struct file_operations fops =
{
        .owner          = THIS_MODULE,
        .read           = test_read,
        .write          = test_write,
        .open           = test_open,
        .unlocked_ioctl = test_ioctl,
        .release        = test_release,
};
```

Create IOCTL command in a Userspace application:

In test_ioctl_app.c define the ioctl commands:

#define WR_VALUE _IOW('a','a',int32_t*)

#define RD_VALUE _IOR('a','b',int32_t*)

Use IOCTL system call in Userspace:

Open device "/dev/test_device".

Perform write and read calls:

Ioctl (fd, WR_VALUE, (int32_t*) and number);

Ioctl (fd, RD_VALUE, (int32_t*) &value

```
#define WR_VALUE _IOW('a','a',int32_t*)
#define RD_VALUE _IOR('a','b',int32_t*)

int main()
{
        int fd;
        int32_t value, number;

        printf("\nOpening Driver : \"/dev/test_device\"\r\n");
        fd = open("/dev/test_device", O_RDWR,0777);
        if(fd < 0) {
                printf("Cannot open device file...\r\n");
                return 0;
        }

        printf("Enter the Value to send\r\n");
        scanf("%d",&number);
        printf("Writing Value to Driver\r\n");
        ioctl(fd, WR_VALUE, (int32_t*) &number);

        printf("Reading Value from Driver\r\n");
        ioctl(fd, RD_VALUE, (int32_t*) &value);
        printf("Recived Value is %d\r\n", value);

        printf("Closing Driver\n");
        close(fd);
}
```

8.4.3 Cross Compile the Device Driver

Make sure you have the Raspberry pi connected to Network.

Connect Raspberry pi via SSH: ssh <pi user name>@<ip address of pi>

Read the kernel version using command: uname –a

- Pi on which demo is run has kernel version 5.5.80-v7+

On your host PC download Raspberry pi kernel source code from official RPI linux kernel repo into Chapter8_Linux_Fundamentals/KDB_Demo/ioctl_example dir:

- git clone https://github.com/raspberrypi/linux --depth=1

On your host PC install the dependencies required cross compile the kernel:

sudo apt install git bc bison flex libssl-dev make libc6-dev libncurses5-dev

Our RPI is armv7 architecture, we need 32bit toolchain:

- sudo apt install crossbuild-essential-armhf

- For 64-bit use:
- sudo apt install crossbuild-essential-arm64

Go the directory linux: cd linux/

- Make mrproper

Compile the Kernel:

- Make KERNEL=kernel7 ARCH=arm CROSS_COMPILE=arm-linux-gnueabihf- bcm2709_defconfig
- Make ARCH=arm CROSS_COMPILE=arm-linux-gnueabihf- zImage modules dtbs -j 5

```
root@pi:/home/pi# uname -a
Linux pi 5.15.80-v7+ #1602 SMP Tue Nov 29 14:42:58 GMT 2022 armv7l GNU/Linux
root@pi:/home/pi#
```

- Compile the driver Run the command: make RPI_build
- Copy to the raspberry pi: scp device_driver_ioctl. ko
  <pi user name>@<ip address of pi>:<path to save driver>/device_driver_ioctl. ko
- Connect to raspberry pi via serial & go to directory where driver is saved.
- Load the driver into system using command:
  sudo insmod device_driver_ioctl. Ko
- To check the driver is loaded run command:
  sudo dmesg
  dmesg is a command
- It prints the message buffer of the kernel.
- Run the application using command:
  sudo. /test_ioctl_app
- Again, check dmesg to see the log.
- To unload the driver, use command: **sudo rmmod device_driver_ioctl.ko**

Dmesg log after loading

```
[ 5678.206105] device_driver_ioctl: loading out-of-tree module taints kernel.
[ 5678.206815] Major = 238 Minor = 0
[ 5678.207748] systemd-journald[127]: Sent WATCHDOG=1 notification.
[ 5678.211653] Device Driver Insert...Done!!!
root@pi:/home/pi/KDB_Demo#
root@pi:/home/pi/KDB_Demo#
```

dmesg is a command

```
root@pi:/home/pi/KDB_Demo# sudo ./test_ioctl_app

Opening Driver : "/dev/test_Dev"
Enter the Value to send
100
Writing Value to Driver
Reading Value from Driver
Recived Value is 100
Closing Driver
root@pi:/home/pi/KDB_Demo#
```

sudo ./test_ioctl_app

Again, check dmesg to see the log.

```
[ 5854.436511] Device File Opened...!!!
[ 5858.124786] Driver Write Value = 100
[ 5858.124842] Driver Value to user = 100
[ 5858.125008] Device File Closed...!!!
```

Dmesg log after running app

Now that our driver is up and running let's put breakpoint inside the code from kdb prompt.
Load the driver again if unloaded in previous step.
Start the kdb session again.
Use breakpoint command:

- **bp <function name>**

We will put breakpoint into **test_ioctl():**

- **bp test_ioctl**

Now run command **go** to resume the kernel execution:
Issue the ioctl call to our driver from the test code.

- **sudo ./test_ioctl_app**

```
root@pi:/home/pi/KDB_Demo# echo g > /proc/sysrq-trigger
[ 6388.559924] sysrq: DEBUG

Entering kdb (current=0x8e636e40, pid 1339) on processor 2 due to Keyboard Entry
[2]kdb> bp test_ioctl
Instruction(i) BP #0 at 0x7f64c000 ([device_driver_ioctl]test_ioctl)
    is enabled   addr at 000000007f64c000, hardtype=0 installed=0

[2]kdb> go
root@pi:/home/pi/KDB_Demo# ./test_ioctl_app

Opening Driver : "/dev/test_Dev"
Enter the Value to send
100
Writing
Entering kdb (current=0x815a8fc0, pid 1934) on processor 3 due to Breakpoint @ 0x7f64c000
[3]kdb>
```

```
[3]kdb> lsmod
Module                  Size  modstruct     Used by
device_driver_ioctl    16384  0x7f64e080    1  (Live) 0x7f64c000 [ ]
nls_utf8               16384  0x7f678040    1  (Live) 0x7f676000 [ ]
cifs                  786432  0x7f623b80    0  (Live) 0x7f58b000 [ ]
```

There are many other tools that can be used to debug application in user space and kernel space.
User space tools:
Printf statements: commonly used method to debug and logging purpose.
Gporf: Performance analysis tool for Linux applications
Valgrind: Debugging and profiling tools for problems like: memory leek, double freeing
Kernel space debugging:
KGDB:
Kernel space source code debugger. It can be used with GDB debugger to debug remote target device. We can use the GDB commands to debug the source code. It can be invoked from the KDB session
Printk ():
It is widely used function to in Linux kernel to debug the kernel source code. It is possible to specify different log levels.
JTAG Debugger
It is a hardware debugger. It is directly connected to JTAG pins on hardware. It is used to access the debug modules inside the target CPU. We can debug the Linux boot sequence, device drivers etc.

9. C Library Functions

Key Learning Outcomes

At the end of this module, you will be able to:

1. Explain the functions associated with the library “stdlib.h”.
2. Explain the various functions associated with date and time operation.
3. State about OOPS.
4. List the advantages of OOPS.
5. Explain the polymorphism, encapsulation, abstraction, and inheritance with suitable example.
6. Explain the GNU libC.
7. Implement syscall functions in the code.

UNIT 9.1: Useful Primitives

Unit Objectives

At the end of this unit, you will be able to:

1. Discuss the use of macro assert()
2. Explain the functions associated with the library “stdlib.h”
3. Explain the various functions associated with date and time operation.
4. Explain the need of system calls.
5. Discuss the types of system calls.

9.1.1 Assert()

Assert is a macro that is used to check specific conditions at runtime. To use it, you must include the header file "assert.h" in the program. Declaration: void assert(int expression). If expression evaluates to TRUE, assert() does nothing. If expression evaluates to FALSE, assert() displays an error message on stderr and aborts program execution.

Example Program: Demo for using the assert() macro.

test_assert.c

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

int main()
{
    int a, b;

    printf("Input two integers to divide\r\n");
    scanf("%d%d", &a, &b);

    // If the condition (b != 0) true,
    // then the program execution will continue.
    // Otherwise it terminates,
    //An error message is displayed on the screen
    assert(b != 0);

    printf("%d/%d = %.2f\n", a, b, a / (float)b);

    return 0;
}
```

```
akshay@akshayv:Chapter9_C_Library_functions$ gcc test_assert.c -o test_assert
akshay@akshayv:Chapter9_C_Library_functions$ ./test_assert
Input two integers to divide
10 0
test_assert: test_assert.c:13: main: Assertion `b != 0' failed.
Aborted (core dumped)
akshay@akshayv:Chapter9_C_Library_functions$ ./test_assert
Input two integers to divide
10 2
10/2 = 5.00
akshay@akshayv:Chapter9_C_Library_functions$
```

9.1.2 C Library Functions - <stdlib.h>

The **stdlib.h** header defines four variable types, several macros, and various functions for performing general functions.

Members of the **stdlib.h** can be classified into the following categories:

- Conversion
- Memory
- Process control
- Sort and search
- Mathematics

Example program : Demo for standard library functions:

test_stdlib.c . After reading and execution of code you be able to understand how to use these standard library functions. To read more about stdlib.h, visit stdlib.h - Linux Man Page

9.1.2.1 Conversion Functions

Function	Syntax	Descripton
atof	double atof(const char *str)	Convert String to Floating-Point
atoi	int atoi(const char *str)	Convert String to Integer
atol	long int atol(const char *str)	Convert String to Long Integer
strtod	double strtod(const char *str, char **endptr)	Convert String to Double
strtol	long int strtol(const char *str, char **endptr, int base)	Convert String to Long Integer
strtoll	long long strtoll(const char *nptr, char **endptr, int base);	Convert String to Long Long
strtoul	unsigned long int strtoul(const char *str, char **endptr, int base)	Convert String to Unsigned Long Integer

9.1.2.2 Dynamic Memory Allocation

Function	Syntax	Description
malloc	(cast-data-type *)malloc(size-in-bytes);	Allocates the specified number of bytes
realloc	void *realloc(void *ptr, size_t size);	Increases or decreases the size of the specified block of memory, moving it if necessary
calloc	(cast-type*)calloc(n, element-size);	Allocates the specified number of bytes and initializes them to zero
free	void free(void *ptr);	Releases the specified block of memory back to the system

9.1.2.3 Process Control

Function	Syntax	Description
abort	void abort(void)	Abort Program
atexit	int atexit(void (*func)(void))	Register Function to be Called at Program Exit
exit	void exit(int status)	Exit from Program
getenv	char *getenv(const char *name)	Get Environment String
system	int system(const char *string)	Perform Operating System Command

9.1.2.4 Binary Search

Function	Syntax	Description
bsearch	void *bsearch(const void *key, const void *base, size_t num_members, size_t size, int (*compare_function) (const void *, const void *)); Parameters or Arguments key The value to search for. base The address in a sorted array to begin the search. num_members The number of elements. size The size of the elements in bytes. compare_function A pointer to a comparison function. **Returns** The bsearch function returns: Negative = key less than the element Zero = key equal to element in array Positive integer = key greater than the element in the array	Binary Search

9.1.2.5 Sort

Function	Syntax	Description
qsort	void qsort(void *base, size_t num_members, size_t size, int (*compare_function) (const void *, const void *)); **Parameters or Arguments** base The array to sort. num_members The number of elements in the array. size The size of the elements in bytes. compare_function A pointer to a comparison function. **Returns** Negative = the first element in the array is less than the second element in the array Zero = the first element in the array is equal to the second element in the array Positive = the first element in the array is greater than the second element in the array	Sort Array

9.1.2.6 Mathematics

Function	Syntax	Description
int abs(int)	int abs(int x);	absolute value of an integer.
long int labs(long int)	long int labs(long int x);	absolute value of a long integer.
div	div_t div(int numerator, int denominator);	integer division (returns quotient and remainder)
ldiv	ldiv_t ldiv(long int numerator, long int denominator);	long integer division (returns quotient and remainder)

div_t and **ldiv_t,** are the return types of the **div** and **ldiv** functions. The standard defines them as:

```
typedef struct {
    int quot, rem;
} div_t;
```

```
typedef struct {
    long int quot, rem;
} ldiv_t;
```

- Example program: Demo for standard library functions: **test_stdlib.c**
- After reading and execution of code you be able to understand how to use these standard library functions.
- To read more about stdlib.h, visit stdlib.h - Linux Man Page

9.1.3 Date and Time Functions

The C date and time functions are a group of functions implementing date and time manipulation operations. They provide support for time acquisition, conversion between date formats, and formatted output to strings. The C date and time operations are defined in the time.h header file.

Time manipulation:

Description	Syntax	Description
difftime	double difftime(time_t time_end, time_t time_beg);	computes the difference in seconds between two time_t values
time	time_t time(time_t *arg);	returns the current time of the system as a time_t value, number of seconds, typically the Unix epoch.
clock	clock_t clock(void);	returns a processor tick count associated with the process
timespec_get (C11)	int timespec_get(struct timespec *ts, int base);	returns a calendar time based on a time base

9.1.3.1 Time Format Conservation

Function	Syntax	Description
asctime	char* asctime(const struct tm* time_ptr);	converts a struct tm object to a textual representation (deprecated)
ctime	char* ctime(const time_t* timer);	converts a time_t value to a textual representation
strftime	size_t strftime(char * str, size_t count, const char * format, const struct tm * time);	converts a struct tm object to custom textual representation
strptime	char *strptime(const char *s, const char *format, struct tm *tm);	converts a string with time information to a struct tm
wcsftime	size_t wcsftime(wchar_t* str, size_t count, const wchar_t* format, tm* time);	converts a struct tm object to custom wide string textual representation
gmtime	struct tm *gmtime (const time_t *timer);	converts a time_t value to calendar time expressed as Coordinated Universal Time
localtime	struct tm *localtime (const time_t *timer);	converts a time_t value to calendar time expressed as local time
mktime	time_t mktime(struct tm *arg);	converts calendar time to a time_t value.

Example program: Demo for date time functions:

test_time_func.c . After reading and execution of code you be able to understand how to use these Date time Functions. To read more about time.h, visit time - Linux manual page.

UNIT 9.2: System Calls

Unit Objectives

At the end of this unit, you will be able to:

1. Explain the need of system calls.
2. Discuss the types of system calls.

9.2.1 System Calls

A system call is a method for a computer program to request a service from the kernel of the operating system on which it is running. The service is generally something that only the kernel has the privilege to do, such as doing I/O. Programmers don't normally need to be concerned with system calls because there are functions in the GNU C Library to do virtually everything that system calls do. The applications run in an area of memory known as user space. A system call connects to the operating system's kernel, which executes in kernel space.

For example: If the program wants to do device-specific input/output operations, then you can just use ioctl().

Why do we need system calls?

It is must require when a file system wants to create or delete a file. Network connections require the system calls to send and receive data packets. If you want to read or write a file, you need to system calls. If you want to access hardware devices, including a printer, scanner, you need a system call. System calls are used to create and manage new processes. For more information visit: syscalls() - Linux manual page.

Types of System Calls:

Process Control	File Management	Device Management	Information Maintenance	Communication
• Fork() • Exit() • Wait()	• Open() • Read() • Write() • Close()	• Ioctl() • Read() • Write()	• Getpid() • Alarm() • Sleep()	• Pipe() • Mmap() • Shmget()

open():

The open() system call allows you to access a file on a file system. It allocates resources to the file and provides a handle that the process may refer to. The return value of open() is a file descriptor, a small, nonnegative integer that is an index to an entry in the process's table of open file descriptors. The file descriptor is used in subsequent system calls: read, write, close

Syntax:
fd = open (file_name, mode, permission);
Parameters:
file_name: Absolute path to file
mode:
O_RDONLY: read only,
O_WRONLY: write only,
O_RDWR: read and write,
O_CREAT: create file if it doesn't exist,
O_EXCL: prevent creation if it already exists
Permission: Notation of traditional Unix permissions
Example:

```
fd = open ("file", O_CREAT | O_RDWR, 0777);
```

Notation of Traditional Unix Permissions		
Symbolic Notation	**Numeric Notation**	**English**
----------	0000	no permissions
-rwx------	0700	read, write, and execute only for owner
-rwxrwx---	0770	read, write, and execute for owner and group
-rwxrwxrwx	0777	read, write, and execute for owner, group and others
---x--x--x	0111	execute
--w--w--w-	0222	write
--wx-wx-wx	0333	write and execute
-r--r--r--	0444	read
-r-xr-xr-x	0555	read and execute
-rw-rw-rw-	0666	read and write
-rwxr-----	0740	owner can read, write, and execute; group can only read; others have no permissions

read():

It is used to obtain data from a file on the file system. It accepts three arguments in general:

- A file descriptor.
- A buffer to store read data.
- The number of bytes to read from the file.

The file descriptor of the file to be read could be used to identify it and open it using **open()** before reading.

Syntax:

length = read(file_descriptor , buffer, max_len);

Example:

n = read(0, buff, 50);

Write():

It is used to write data from a user buffer to a device like a file. This system call is one way for a program to generate data. It takes three arguments in general:

- A file descriptor.
- A pointer to the buffer in which data is saved.
- The number of bytes to be written from the buffer.

Syntax:

length = write(file_descriptor , buffer, len);

Example:

n = write(fd, "Hello world!", 12

Close():

It is used to end file system access. When this system call is invoked, it signifies that the program no longer requires the file, and the buffers are flushed, the file information is altered, and the file resources are de-allocated as a result.

Syntax:

int close(int fd);

fork():

fork() creates a new process by duplicating the calling process. The new process is referred to as the child process. The calling process is referred to as the parent process. The child process and the parent process run in separate memory spaces.

fork() return the following values:

- **Negative value -** it represents the creation of the child process was unsuccessful.
- **Zero -** it represents a new child process is created.
- **Positive value -** The process ID of the child process to the parent. The returned process ID is type pid_t defined in sys/types.h. Usually, the process ID is an integer.

After a new child process is created, both processes will execute the next instruction following the fork() system call. A child process uses the same pc(program counter), same CPU registers, and same open files which use in the parent process.

Consider the Example 1:

Where one of the outputs came from the parent process and the other one from the child process. Simply, we can tell that the result is 2 power of n, where n is the number of fork() system calls.

onsider the Example 2:

The number of times 'hello' is printed is equal to number of processes created. Total Number of Processes = 2^n, where n is number of fork system calls.

So here n = 3, $2^3 = 8$

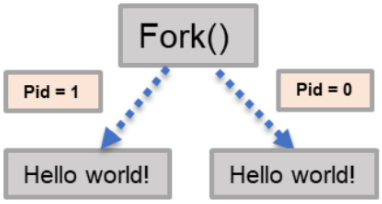
The main process: **P0**

Processes created by the 1st fork: **P1**

Processes created by the 2nd fork: **P2, P3**

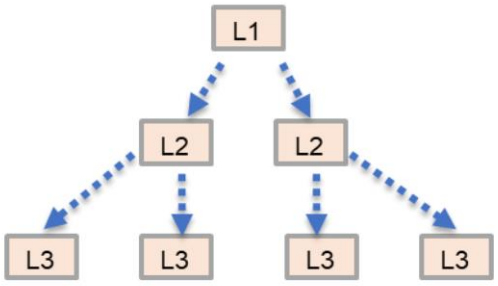
Processes created by the 3rd fork: **P4, P5, P6, P7**

Example Program: **test_fork_call.c**



Output:
Hello world!
Hello world!

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
int main()
{
   /* fork a process */
   fork();
  /* the child and parent will execute every line of code after the fork (each separately)*/
   printf("Hello world!\n");
   return 0;
}
```



Output:

Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
#include <stdlib.h>
int main()
{
   fork(); // L1
   fork(); // L2
   fork(); // L3
   printf("Hello world!\n");
   return 0;
}
```

9.2.2 File Management System Calls

File Descriptor:

In Unix like computer OS, a file descriptor is a process-unique identifier for a file or input/output resource. Each Unix process (except perhaps daemons) should have three standard POSIX file descriptors, corresponding to the three standard streams:

Integer value	Name	<unistd.h> symbolic constant	<stdio.h> file stream
0	Standard input	STDIN_FILENO	stdin
1	Standard output	STDOUT_FILENO	stdout
2	Standard error	STDERR_FILENO	stderr

Here are some **system calls for file management** which operate on file descriptors:

- **open ()**
- **read ()**
- **write ()**
- **close ()**

Example Program:

test_file_system_call.c

```
akshay@akshayv:Chapter9_C_Library_functions$ gcc test_file_system_call.c -o test_file_system_call
akshay@akshayv:Chapter9_C_Library_functions$ ./test_file_system_call
Enter text to write in the file:
This tutorial we will read data from keyboard input and write to file.txt!!
This tutorial we will read data from keyboard input and write to file.txt!!

akshay@akshayv:Chapter9_C_Library_functions$ cat file.txt
This tutorial we will read data from keyboard input and write to file.txt!!
akshay@akshayv:Chapter9_C_Library_functions$
```

New File "**file.txt**" created after script execution

9.2.2 System Calls: ioctl()

I**octl()** is an abbreviation of input/output control. It is a system call for device-specific input/output operations and other operations which cannot be expressed by regular system calls. It takes a parameter specifying a request code; the effect of a call depends completely on the request code. Request codes are often device-specific. For instance, a **CD-ROM device driver** which can instruct a physical device to **eject a disc** would provide an **ioctl request** code to do so.

Steps involved in IOCTL:

- Create IOCTL command in the driver
- Write IOCTL function in the driver
- Create IOCTL command in a Userspace application
- Use the IOCTL system call in a Userspace

Example Program:

device_driver_ioctl.c : Simple Device driver with IOCTL

test_ioctl_app.c : Test app which usees IOCTL request.

```
akshay@akshayv:ioctl_example$ tree
.
├── device_driver_ioctl.c
├── Makefile
└── test_ioctl_app.c
```

Define the ioctl command:

#define "Name of IOCTL" _IOx(num1, num2, argument type)

where **IOx** can be :

"**IO**": an ioctl with no parameters

"**IOW**": an ioctl with write parameters (copy_from_user)

"**IOR**": an ioctl with read parameters (copy_to_user)

"**IOWR**": an ioctl with both write and read parameters

num1: is a unique number or character that will differentiate our set of ioctl calls from the other ioctl calls. sometimes the major number for the device is used here.

num2: is the number that is assigned to the ioctl. This is used to differentiate the commands from one another.

argument type: Type of data.

Command example used in our driver:

```
#define WR_VALUE _IOW('a','a',int32_t*)
#define RD_VALUE _IOR('a','b',int32_t*)
```

Write IOCTL function in the driver

We need to add the ioctl function to our driver. The prototype is as shown in image:

"file": the file pointer to the file that was passed by the application.

"cmd" : the ioctl command that was called from the userspace.

"arg": are the arguments passed from the userspace.

Within function we implement all the commands we defined. To inform kernel that ioctl call is implemented in test_ioctl():

We make **fops** poiter **unlocked_ioctl** point to **test_ioctl.**

```c
static long test_ioctl(struct file *file, unsigned int cmd, unsigned long arg)
{
        switch(cmd) {
                case WR_VALUE:
                        if( copy_from_user(&value ,(int32_t*) arg, sizeof(value)) )
                        {
                                pr_err("Data Write : Err!\n");
                        }
                        pr_info("Driver Write Value = %d\n", value);
                        break;
                case RD_VALUE:
                        if( copy_to_user((int32_t*) arg, &value, sizeof(value)) )
                        {
                                pr_err("Data Read : Err!\n");
                        }
                        pr_info("Driver Value to user = %d\n", value);
                        break;
                default:
                        pr_info("Default\n");
                        break;
        }
        return 0;
}
```

```c
/*
** File operation sturcture
*/
static struct file_operations fops =
{
        .owner          = THIS_MODULE,
        .read           = test_read,
        .write          = test_write,
        .open           = test_open,
        .unlocked_ioctl = test_ioctl,
        .release        = test_release,
};
```

Create IOCTL command in a Userspace application:

In test_ioctl_app.c define the ioctl commands:

#define WR_VALUE _IOW('a','a',int32_t*)

#define RD_VALUE _IOR('a','b',int32_t*)

Use IOCTL system call in Userspace:

Open device **"/dev/test_device".**

Perform write and read calls:

ioctl(fd, WR_VALUE, (int32_t*) &number);

ioctl(fd, RD_VALUE, (int32_t*) &value);

```c
#define WR_VALUE _IOW('a','a',int32_t*)
#define RD_VALUE _IOR('a','b',int32_t*)

int main()
{
        int fd;
        int32_t value, number;

        printf("\nOpening Driver : \"/dev/test_device\"\r\n");
        fd = open("/dev/test_device", O_RDWR,0777);
        if(fd < 0) {
                printf("Cannot open device file...\r\n");
                return 0;
        }

        printf("Enter the Value to send\r\n");
        scanf("%d",&number);
        printf("Writing Value to Driver\r\n");
        ioctl(fd, WR_VALUE, (int32_t*) &number);

        printf("Reading Value from Driver\r\n");
        ioctl(fd, RD_VALUE, (int32_t*) &value);
        printf("Recived Value is %d\r\n", value);

        printf("Closing Driver\n");
        close(fd);
}
```

Build the driver by command:
sudo make

This will build the driver device_driver_ioctl.ko

Build the application code with command:

gcc -o test_ioctl_app test_ioctl_app.c

Load the driver into system using command:

sudo insmod device_driver_ioctl.ko

To check the driver is loaded run command:

sudo dmesg

dmesg is a command that prints the message buffer of the kernel.

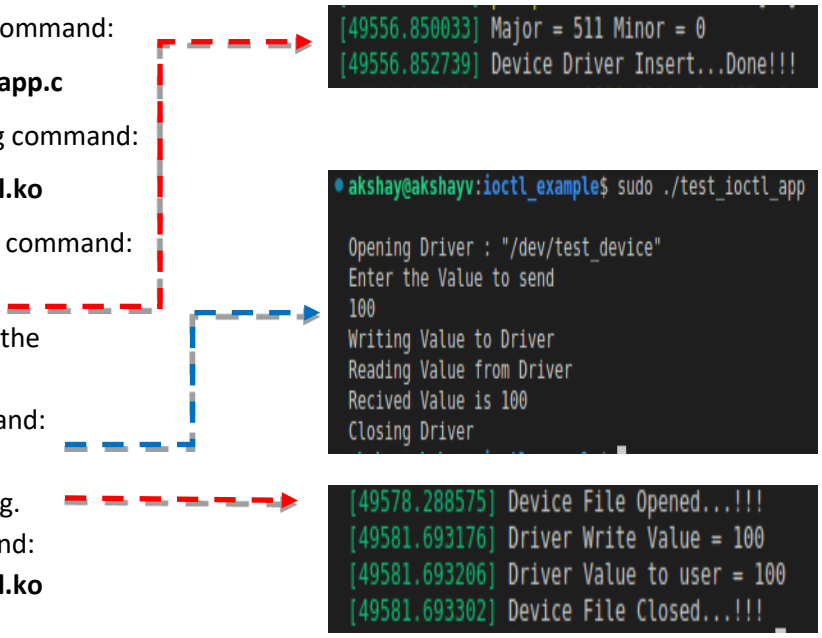
Run the application using command:

sudo ./test_ioctl_app

Again, check dmesg to see the log.

To unload the driver use command:

sudo rmmod device_driver_ioctl.ko



UNIT 9.3: OOPS Concept

Unit Objectives

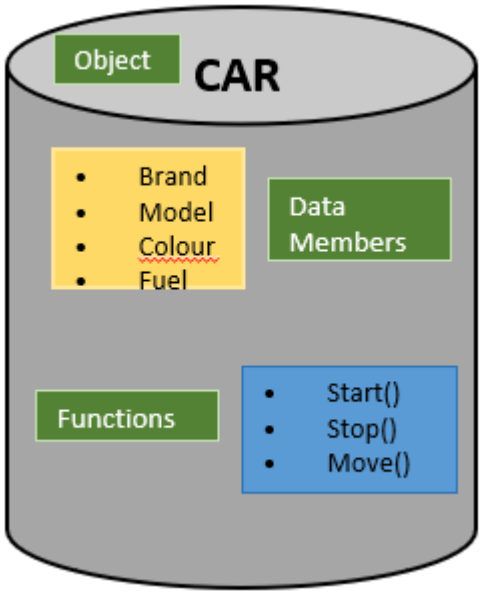
At the end of this unit, you will be able to:

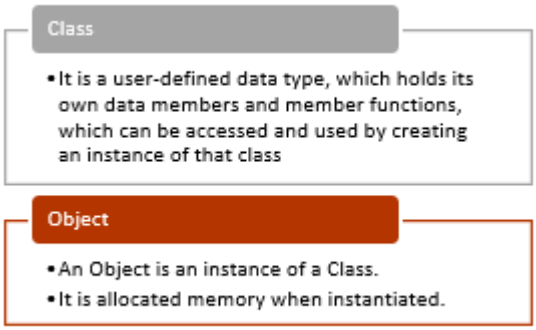
1. State about OOPS.
2. List the advantages of OOPS.
3. Explain the polymorphism with suitable example.
4. Explain the encapsulation with suitable example.
5. Explain the abstraction with suitable example.
6. Explain the inheritance with suitable example.

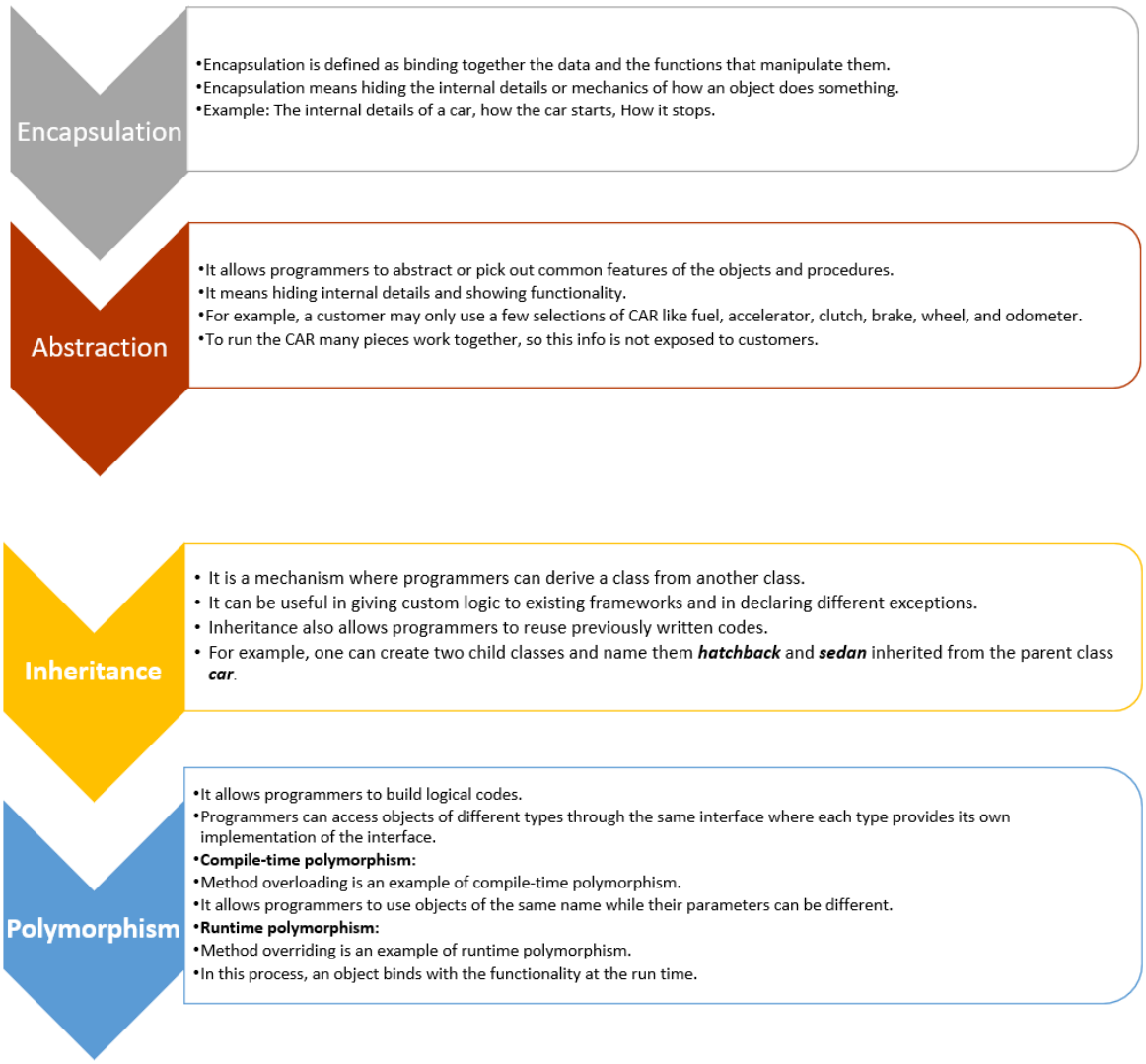
9.3.1 OOPS Concept

OOPS is a programming model based on the concept of objects and classes. In this model, programmers define the functions that can be applicable to the data structures and their data type. Object-oriented programming turns data structure into an object, including both data and functions. It encourages the reusing of these objects in the same and other programs as well. These days, most major software development is performed using OOP. With OOP, instead of writing a program, you create classes. A class contains both data and functions. When you want to create something in memory, you create an object, which is an instance of that class.

For example: We create class CAR, which holds data and functions related to CAR. When the program want to create a CAR in memory, then a new object of CAR is created.







Advantages of OOPS:

Modularity for easier troubleshooting:

You know exactly where to look when something goes wrong. “Oh, the car object broke down? The problem must be in the Car class!”

Reuse of code through inheritance:

Create one generic class (Car), and then define the subclasses (Sedan , Hatchback, SUV etc.)

Flexibility through polymorphism

Because a single function can shape-shift to adapt to whichever class it’s in, you could create one function in the parent Add class called sum() -- not sum2elements() or sum3elements() but just sum().

Polymorphism

```
Int main()
{
Sum1 = sum(2,5);
Sum2 = sum(20,23,1);
}
```

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

```
Int sum(int a, int b, int c)
{
return a+b+c;
}
```

OOPS concept in C programming:

Although the fundamental OOP concepts have been traditionally associated with object-oriented languages, such as C++, or Java. You can implement them in almost any programming language including portable, standard-compliant C (ISO-C90 Standard). One good reference for implementation of OOPS concepts can be found here OOP-in-C. Consider we have an application that employs 2D shapes to be render on Graphic-LCD. We can come up with our classes.

Encapsulation in C:

Attributes of class are defined with C struct

Operations of class are defined as C functions.

Consider the Application: Chapter9_C_Library_functions/OOPS_in_C/encapsulation.

Shape class

```c
#ifndef SHAPE_H
#define SHAPE_H

#include <stdint.h>

/* Shape's attributes... */
typedef struct {
    int16_t x; /* x-coordinate of Shape's position */
    int16_t y; /* y-coordinate of Shape's position */
} Shape;

/* Shape's operations (Shape's interface)... */
void Shape_ctor(Shape * const me, int16_t x, int16_t y);
void Shape_moveBy(Shape * const me, int16_t dx, int16_t dy);
int16_t Shape_getX(Shape const * const me);
int16_t Shape_getY(Shape const * const me);

#endif /* SHAPE_H */
```

shape.h

```c
#include "shape.h" /* Shape class interface */

/* constructor implementation */
void Shape_ctor(Shape * const me, int16_t x, int16_t y) {
    me->x = x;
    me->y = y;
}

/* move-by operation implementation */
void Shape_moveBy(Shape * const me, int16_t dx, int16_t dy) {
    me->x += dx;
    me->y += dy;
}

/* "getter" operations implementation */
int16_t Shape_getX(Shape const * const me) {
    return me->x;
}
int16_t Shape_getY(Shape const * const me) {
    return me->y;
}
```

shape.c

OOPS concept in C Programming:

Encapsulation in C:

- Compile the code: gcc main.c shape.c -o test_encapsulation
- Shape_getX() and Shape_getY() : Get the value of x and y attributes
- Shape_moveBy() : will add dx and dy to x and y attributes.
- This will move the 2D shape on graph according to new x and y attrbutes.

```
akshay@akshayv:encapsulation$ gcc main.c shape.c -o test_encapsulation
akshay@akshayv:encapsulation$ ./test_encapsulation
Shape s1(x=0,y=1)
Shape s2(x=-1,y=2)
Shape s1(x=2,y=-3)
Shape s2(x=0,y=0)
akshay@akshayv:encapsulation$
```

main.c

```c
#include "shape.h"  /* Shape class interface */
#include <stdio.h>  /* for printf() */

int main() {
    Shape s1, s2; /* multiple instances of Shape */

    Shape_ctor(&s1, 0, 1);
    Shape_ctor(&s2, -1, 2);

    printf("Shape s1(x=%d,y=%d)\n", Shape_getX(&s1), Shape_getY(&s1));
    printf("Shape s2(x=%d,y=%d)\n", Shape_getX(&s2), Shape_getY(&s2));

    Shape_moveBy(&s1, 2, -4);
    Shape_moveBy(&s2, 1, -2);

    printf("Shape s1(x=%d,y=%d)\n", Shape_getX(&s1), Shape_getY(&s1));
    printf("Shape s2(x=%d,y=%d)\n", Shape_getX(&s2), Shape_getY(&s2));

    return 0;
}
```

Inheritance:

Instead of creating the new class for **Rectangle** from scratch. We can inherit what most common from the **Shape** class and add only what is different for defining rectangles. Consider the Application : **Chapter9_C_Library_functions/OOPS_in_C/inheritance**

Rectangle Class

rect.h

```c
#ifndef RECT_H
#define RECT_H

#include "shape.h" /* the base class interface */

/* Rectangle's attributes... */
typedef struct {
    Shape super; /* <== inherits Shape */

    /* attributes added by this subclass... */
    uint16_t width;
    uint16_t height;
} Rectangle;

/* constructor prototype */
void Rectangle_ctor(Rectangle * const me, int16_t x, int16_t y,
                    uint16_t width, uint16_t height);

#endif /* RECT_H */
```

rect.c

```c
#include "rect.h"

/* constructor implementation */
void Rectangle_ctor(Rectangle * const me, int16_t x, int16_t y,
                    uint16_t width, uint16_t height)
{
    /* first call superclass' ctor */
    Shape_ctor(&me->super, x, y);

    /* next, you initialize the attributes added by this subclass... */
    me->width = width;
    me->height = height;
}
```

Output observations:

You can see how to instantiate the rectangle: **Rectangle_ctor()**. Using **Shape_moveBy()** we can change the shape on x and y coordinates

```
akshay@akshayv:inheritance$ gcc main.c shape.c rect.c -o test_inheritance
akshay@akshayv:inheritance$ ./test_inheritance
  Rect r1(x=0,y=2,width=10,height=15)
  Rect r2(x=-1,y=3,width=5,height=8)
  Rect r1(x=-2,y=5,width=10,height=15)
  Rect r2(x=1,y=2,width=5,height=8)
akshay@akshayv:inheritance$
```

main.c

```c
#include "rect.h"  /* Rectangle class interface */
#include <stdio.h>  /* for printf() */

int main() {
    Rectangle r1, r2; /* multiple instances of Rect */

    /* instantiate rectangles... */
    Rectangle_ctor(&r1, 0, 2, 10, 15);
    Rectangle_ctor(&r2, -1, 3, 5, 8);

    printf("Rect r1(x=%d,y=%d,width=%d,height=%d)\n",
           Shape_getX(&r1.super), Shape_getY(&r1.super),
           r1.width, r1.height);
    printf("Rect r2(x=%d,y=%d,width=%d,height=%d)\n",
           Shape_getX(&r2.super), Shape_getY(&r2.super),
           r2.width, r2.height);

    /* re-use inherited function from the superclass Shape... */
    Shape_moveBy((Shape *)&r1, -2, 3);
    Shape_moveBy(&r2.super, 2, -1);

    printf("Rect r1(x=%d,y=%d,width=%d,height=%d)\n",
           Shape_getX(&r1.super), Shape_getY(&r1.super),
           r1.width, r1.height);
    printf("Rect r2(x=%d,y=%d,width=%d,height=%d)\n",
           Shape_getX(&r2.super), Shape_getY(&r2.super),
           r2.width, r2.height);

    return 0;
}
```

OOPS concept in C Programming:

Polymorphism (Virtual Function) in C:

Consider the Shape class we saw before. We can provide more useful operations:

Area (): To calculate area of shape

Draw (): to draw the shape on screen when integrated with G-LCD. But Shape class cannot provide the implementation of different shapes like circle, rectangle. But it can provide basic interface for those operations:

Shape_area() , **Shape_draw()**.

Consider example:

Chapter9_C_Library_functions/OOPS_in_C/polymorphism

We define the structure Shape with Shape's virtual pointer. This interface will allow us to write generic code to manipulate the shapes.

```c
#include <stdint.h>

/* Shape's attributes... */
struct ShapeVtbl; /* forward declaration */
typedef struct {
    struct ShapeVtbl const *vptr; /* <== Shape's Virtual Pointer */
    int16_t x; /* x-coordinate of Shape's position */
    int16_t y; /* y-coordinate of Shape's position */
} Shape;

/* Shape's virtual table */
struct ShapeVtbl {
    uint32_t (*area)(Shape const * const me);
    void (*draw)(Shape const * const me);
};

/* Shape's operations (Shape's interface)... */
void Shape_ctor(Shape * const me, int16_t x, int16_t y);
void Shape_moveBy(Shape * const me, int16_t dx, int16_t dy);
int16_t Shape_getX(Shape const * const me);
int16_t Shape_getY(Shape const * const me);

static inline uint32_t Shape_area(Shape const * const me) {
    return (*me->vptr->area)(me);
}

static inline void Shape_draw(Shape const * const me) {
    (*me->vptr->draw)(me);
}

/* generic operations on collections of Shapes */
Shape const *largestShape(Shape const *shapes[], uint32_t nShapes);
void drawAllShapes(Shape const *shapes[], uint32_t nShapes);

#endif /* SHAPE_H */
```

shape.h

Virtual functions: Shape_area() & Shape_draw() can have different implementation for subclass rectangle and Circle:

Rect subclass of **Shape**:

Area calculation is: width * height

Circle subclass of **Shape**:

Area Calculation is: pi * radius^2

This means the virtual function call cannot be resolved at link type as we do for normal function in C. The actual version of function call will depend on the type of object Rect or Circle.

- **ShapeVtbl**: Is table of function pointers corresponding to virtual functions.
- Vptr is pointer to Virtual table of class
- This pointer must be present for each class.
- Vptr is declared as pointer to an immutable object see const keyword in front of it.
- Vptr should not change and is allocated in ROM,

Setting vtpr in constructor:

The vptr must be initializes to corresponding virtual table in every instance of class. In c you need to initialize the vptr in Shape constructor as implemented in shape.c in **Shape_ctor()**. The vtbl is initialized to function pointers **Shape_area_()** & **Shape_draw_().** If class cannot provide implementation of its virtual functions, as it is an abstract class of Shape, the implementations should assert internally.

```c
/* Shape class implementations of its virtual functions... */
static uint32_t Shape_area_(Shape const * const me) {
    assert(0); /* purely-virtual function should never be called */
    return 0U; /* to avoid compiler warnings */
}

static void Shape_draw_(Shape const * const me) {
    assert(0); /* purely-virtual function should never be called */
}
```

shape.c

```c
#include "shape.h"
#include <assert.h>

/* Shape's prototypes of its virtual functions */
static uint32_t Shape_area_(Shape const * const me);
static void Shape_draw_(Shape const * const me);

/* constructor */
void Shape_ctor(Shape * const me, int16_t x, int16_t y) {
    static struct ShapeVtbl const vtbl = { /* vtbl of the Shape class */
        &Shape_area_,
        &Shape_draw_
    };
    me->vptr = &vtbl; /* "hook" the vptr to the vtbl */
    me->x = x;
    me->y = y;
}
```

shape.c

Inheriting the vtbl and Overriding vptr

vptr is inherited automatically by all subclasses. **vptr** needs to be re-assigned to the **vtbl** of specific subclas. This re-assignment must happen inside subclass constructor. For example consider :**rect.h** subclasses of **Shape**

```c
#ifndef RECT_H
#define RECT_H

#include "shape.h" /* the base class interface */

/* Rectangle's attributes... */
typedef struct {
    Shape super; /* <== inherits Shape */

    /* attributes added by this subclass... */
    uint16_t width;
    uint16_t height;
} Rectangle;

/* constructor prototype */
void Rectangle_ctor(Rectangle * const me, int16_t x, int16_t y,
                    uint16_t width, uint16_t height);

#endif /* RECT_H */
```

rect.h

rect.c

```c
/* Rectangle's prototypes of its virtual functions */
/* NOTE: the "me" pointer has the type of the superclass to fit the vtable */
static uint32_t Rectangle_area_(Shape const * const me);
static void Rectangle_draw_(Shape const * const me);

/* constructor */
void Rectangle_ctor(Rectangle * const me, int16_t x, int16_t y,
                    uint16_t width, uint16_t height)
{
    static struct ShapeVtbl const vtbl = { /* vtbl of the Rectangle class */
        &Rectangle_area_,
        &Rectangle_draw_
    };
    Shape_ctor(&me->super, x, y); /* call the superclass' ctor */
    me->super.vptr = &vtbl; /* override the vptr */
    me->width = width;
    me->height = height;
}

/* Rectangle's class implementations of its virtual functions... */
static uint32_t Rectangle_area_(Shape const * const me) {
    Rectangle const * const me_ = (Rectangle const *)me; /* explicit downcast */
    return (uint32_t)me_->width * (uint32_t)me_->height;
}

static void Rectangle_draw_(Shape const * const me) {
    Rectangle const * const me_ = (Rectangle const *)me; /* explicit downcast */
    printf("Rectangle_draw_(x=%d,y=%d,width=%d,height=%d)\n",
           Shape_getX(me), Shape_getY(me), me_->width, me_->height);
}
```

- For example consider: **circle.h** subclasses of **Shape**
- **Shape_ctor()** is called first to initialize the **me->super** member inherited from **Shape.**
- This sets the vptr to point Shape vtbl.
- Next statement vptr is overridden where it is assigned to Circle's vtbl

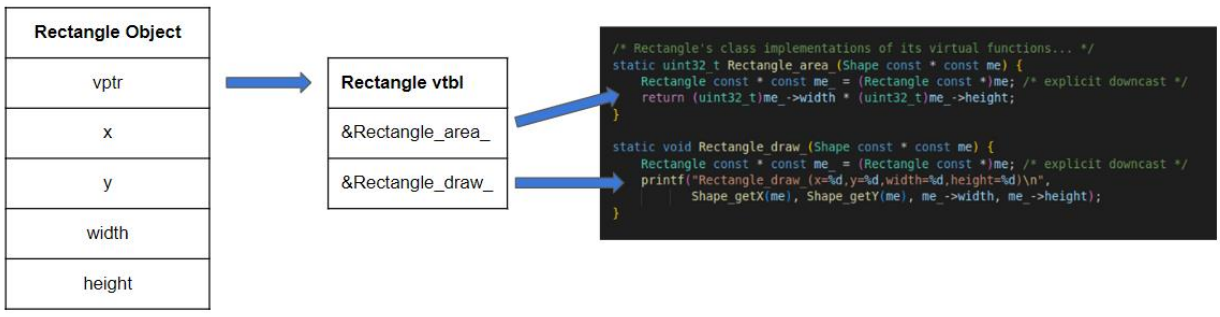
circle.h

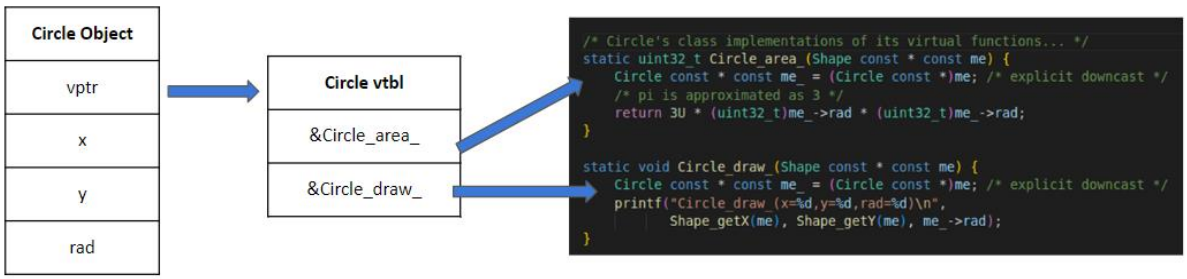
```c
#ifndef CIRCLE_H
#define CIRCLE_H

#include "shape.h" /* the base class interface */

typedef struct {
    Shape super; /* <== inherits Shape */

    /* attributes added by this subclass... */
    uint16_t rad;
} Circle;

/* constructor */
void Circle_ctor(Circle * const me, int16_t x, int16_t y,
                 uint16_t rad);

#endif /* CIRCLE_H */
```

circle.c

```c
/* NOTE: the "me" pointer has the type of the superclass to fit the vtable */
static uint32_t Circle_area_(Shape const * const me);
static void Circle_draw_(Shape const * const me);

/* constructor */
void Circle_ctor(Circle * const me, int16_t x, int16_t y,
                 uint16_t rad)
{
    static struct ShapeVtbl const vtbl = { /* vtbl of the Circle class */
        &Circle_area_,
        &Circle_draw_
    };
    Shape_ctor(&me->super, x, y); /* call the superclass' ctor */
    me->super.vptr = &vtbl; /* override the vptr */
    me->rad = rad;
}

/* Circle's class implementations of its virtual functions... */
static uint32_t Circle_area_(Shape const * const me) {
    Circle const * const me_ = (Circle const *)me; /* explicit downcast */
    /* pi is approximated as 3 */
    return 3U * (uint32_t)me_->rad * (uint32_t)me_->rad;
}

static void Circle_draw_(Shape const * const me) {
    Circle const * const me_ = (Circle const *)me; /* explicit downcast */
    printf("Circle_draw_(x=%d,y=%d,rad=%d)\n",
           Shape_getX(me), Shape_getY(me), me_->rad);
}
```

Virtual Call Mechanism for Rectangle and Circle Class:





Consider main.c :

main.c

```c
#include "rect.h"   /* Rectangle class interface */
#include "circle.h" /* Circle class interface */
#include <stdio.h>  /* for printf() */

int main() {
    Rectangle r1, r2; /* multiple instances of Rectangle */
    Circle    c1, c2; /* multiple instances of Circle */
    Shape const *shapes[] = { /* collection of shapes */
        &c1.super,
        &r2.super,
        &c2.super,
        &r1.super
    };
    Shape const *s;

    /* instantiate rectangles... */
    Rectangle_ctor(&r1, 0, 2, 10, 15);
    Rectangle_ctor(&r2, -1, 3, 5, 8);

    /* instantiate circles... */
    Circle_ctor(&c1, 1, -2, 12);
    Circle_ctor(&c2, 1, -3, 6);

    s = largestShape(shapes, sizeof(shapes)/sizeof(shapes[0]));
    printf("largetsShape s(x=%d,y=%d)\n",
           Shape_getX(s), Shape_getY(s));

    drawAllShapes(shapes, sizeof(shapes)/sizeof(shapes[0]));

    return 0;
}
```

Collection of different shapes

Instantiate the attributes of Rect & Circle

shape.c

```c
/* the following code finds the largest-area shape in the collection */
Shape const *largestShape(Shape const *shapes[], uint32_t nShapes) {
    Shape const *s = (Shape *)0;
    uint32_t max = 0U;
    uint32_t i;
    for (i = 0U; i < nShapes; ++i) {
        uint32_t area = Shape_area(shapes[i]); /* virtual call */
        if (area > max) {
            max = area;
            s = shapes[i];
        }
    }
    return s; /* the largest shape in the array shapes[] */
}

/* The following code will draw all Shapes on the screen */
void drawAllShapes(Shape const *shapes[], uint32_t nShapes) {
    uint32_t i;
    for (i = 0U; i < nShapes; ++i) {
        Shape_draw(shapes[i]); /* virtual call */
    }
}
```

This run's a loop on array of *shapes[] and finds max area by using virtual function of each subclass.

Returns shape with largest area.

This run's a loop on array of *shapes[] & calls the virtual function to draw shape for each subclass.

```
akshay@akshayv:polymorphism$ gcc main.c shape.c circle.c rect.c -o test_polymorphism
akshay@akshayv:polymorphism$ ./test_polymorphism
largetsShape s(x=1,y=-2)
Circle_draw_(x=1,y=-2,rad=12)
Rectangle_draw_(x=-1,y=3,width=5,height=8)
Circle_draw_(x=1,y=-3,rad=6)
Rectangle_draw_(x=0,y=2,width=10,height=15)
akshay@akshayv:polymorphism$
```

Largest shape c1

Draw All shapes

Linux Open Source Code:

QP/C

The OOPS concepts are implemented in the QP/C real-time framework.

QP™/C Real-Time Embedded Framework (RTEF) is a lightweight implementation of the Active Object model of computation specifically tailored for real-time embedded (RTE) systems. QP is both a software infrastructure for building applications consisting of active objects (actors) and a runtime environment for executing the active objects in a deterministic fashion. QP™/C is licensed under the sustainable dual licensing model, in which both the open-source software distribution mechanism and traditional closed source software distribution models are combined.

UNIT 9.4 Referring to Implementations

Unit Objectives

At the end of this unit, you will be able to:

1. Explain the GNU libC
2. Implement syscall functions in the code.

9.5.1 GNU C Library (glibc) project

The GNU C Library is used as *the* C library in the GNU system and in GNU/Linux systems, as well as many other systems that use Linux as the kernel. The GNU C Library is primarily designed to be a portable and high performance C library. The GNU C Library webpage is at http://www.gnu.org/software/libc/ We can study the implementation of libraries in glibc just download the source code. Clone the latest glibc 2.36 source code: git clone https://sourceware.org/git/glibc.git

cd glibc

git checkout release/2.36/master

```
akshay@akshayv:glibc$ ls
abi-tags        COPYING         gshadow          LICENSES      mathvec       resource       sunrpc
aclocal.m4      COPYING.LIB     hesiod           locale        misc          rt             support
argp            crypt           htl              localedata    NEWS          Rules          sysdeps
assert          csu             hurd             login         nis           scripts        sysvipc
benchtests      ctype           iconv            mach          nptl          setjmp         termios
bits            debug           iconvdata        MAINTAINERS   nptl_db       shadow         test-skeleton.c
catgets         dirent          include          Makeconfig    nscd          SHARED-FILES   time
ChangeLog.old   dlfcn           inet             Makefile      nss           shlib-versions timezone
config.h.in     elf             INSTALL          Makefile.help o-iterator.mk signal         version.h
config.make.in  extra-lib.mk    intl             Makefile.in   po            socket         wcsmbs
configure       gen-locales.mk  io               Makerules     posix         soft-fp        wctype
configure.ac    gmon            libc-abis        malloc        pwd           stdio-common
conform         gnulib          libio            manual        README        stdlib
CONTRIBUTED-BY  grp             libof-iterator.mk math         resolv        string
akshay@akshayv:glibc$
```

You can find the implementation of all the lbrary function in this repo.

For eg : **stdlib**

We studied some stdlib.h library functions in this chapter. In the **glibc/stdlib** directory we can find the implementation of those functions.

For eg:

- Mathematical function **div()** in **div.c**
- **Abs()** Get Absolute value of number in **abs.c**

```c
#include <stdlib.h>

#undef  abs

/* Return the absolute value of I.  */
int
abs (int i)
{
  return i < 0 ? -i : i;
}
```

```c
#include <stdlib.h>

/* Return the `div_t' representation of NUMER over DENOM.  */
div_t
div (int numer, int denom)
{
  div_t result;

  result.quot = numer / denom;
  result.rem = numer % denom;

  return result;
}
```

9.5.2 Linux Syscall Implementation

In order to read the System calls implementation we should look into Linux Kernel Sources.
Linux Kernel Source Can be found on github: https://github.com/torvalds/linux System calls are not stored in just one particular location in whole kernel tree.
This is because different syscalls can refer to different parts of system :

- Process Management – fork(), exit(), exec()
- File management – open(), read(), write(), close()
- Device Management - ioctl()
- Information Maintenance – getpid(), alarm(), sleep()

You can read SYSCALL_DEFINE[0-6] macro.
For example : fs/ioctl.c

```
SYSCALL_DEFINE3(ioctl, unsigned int, fd, unsigned int, cmd, unsigned long, arg)
{
        struct fd f = fdget(fd);
        int error;

        if (!f.file)
                return -EBADF;

        error = security_file_ioctl(f.file, cmd, arg);
        if (error)
                goto out;

        error = do_vfs_ioctl(f.file, fd, cmd, arg);
        if (error == -ENOIOCTLCMD)
                error = vfs_ioctl(f.file, cmd, arg);

out:
        fdput(f);
        return error;
}
```

For example : kernel/fork.c
fork()

```
#ifdef __ARCH_WANT_SYS_FORK
SYSCALL_DEFINE0(fork)
{
#ifdef CONFIG_MMU
        struct kernel_clone_args args = {
                .exit_signal = SIGCHLD,
        };

        return kernel_clone(&args);
#else
        /* can not support in nommu mode */
        return -EINVAL;
#endif
}
#endif
```

For example : kernel/sys.c
getpid()

```c
/**
 * sys_getpid - return the thread group id of the current process
 *
 * Note, despite the name, this returns the tgid not the pid.  The tgid and
 * the pid are identical unless CLONE_THREAD was specified on clone() in
 * which case the tgid is the same in all threads of the same group.
 *
 * This is SMP safe as current->tgid does not change.
 */
SYSCALL_DEFINE0(getpid)
{
        return task_tgid_vnr(current);
}
```

For example : **fs/open.c**

open()

```c
SYSCALL_DEFINE3(open, const char __user *, filename, int, flags, umode_t, mode)
{
        if (force_o_largefile())
                flags |= O_LARGEFILE;
        return do_sys_open(AT_FDCWD, filename, flags, mode);
}
```

10. Coding Practices

Unit 10.1 - Coding Guidelines

Unit 10.2 - Secure and Safe Coding

Unit 10.3 - Develop Optimal Code

Key Learning Outcomes

At the end of this module, you will be able to:

1. Implement the coding guidelines while developing a code
2. Avoid the pitfalls/limitation of various concepts in C while developing a code
3. Write an optimal code in terms of memory and performance.

UNIT 10.1 Coding Guidelines

Unit Objectives

At the end of this unit, you will be able to:

1. Implement the coding guidelines while developing a code

10.1.1 Coding Standards

Good software development organizations want their programmers to maintain to some well-defined and standard style of coding called coding standards. They usually make their own coding standards and guidelines depending on what suits their organization best and based on the types of software they develop. It is very important for the programmers to maintain the coding standards otherwise the code will be rejected during code review. Coding best practices and standards vary depending on the industry a specific product is being built for. The standards required for coding software for luxury automobiles will differ from those for coding software for gaming.

Advantages of implementing Coding Standards:

- Offers uniformity to the code created by different engineers.
- Enables the creation of reusable code.
- Makes it easier to detect errors.
- Make code simpler, more readable, and easier to maintain.
- Boost programmer efficiency and generates faster results.

Some Coding Standards:

- CERT , MISRA. Barr Group's Embedded C Coding Standard

10.1.2 Key Considerations While Developing Code

Naming Conventions:

Modules:

All module names shall consist entirely of lowercase letters, numbers, and underscores. No spaces shall appear within the module's header and source file names. All module names shall be unique in their first 8 characters and end with suffices .h and .c for the header and source file names, respectively. No module's header file name shall share the name of a header file from the C Standard Library or C++ Standard Library. For example, modules shall not be named "**stdio**" or "**math**". Any module containing a main() function shall have the word "main" as part of its source file name eg : **app_main.c**

Data Type Rules:

The names of all new data types, including structures, unions, and enumerations, shall consist only of lowercase characters and internal underscores and end with '_t'. All new structures, unions, and enumerations shall be named via a typedef. The name of all public data types shall be prefixed with their module name and an underscore.

```
typedef struct
{
    uint16_t count;
    uint16_t max_count;
    uint16_t _unused;
    uint16_t control;
} timer_reg_t;
```

Variable Rules:

No variable shall have a name that is a keyword of C, C++, or any other well-known extension of the C programming language, including specifically K&R C and C99. Restricted names include **interrupt**, **inline**, **restrict**, **class**, **true**, **false**, **public**, **private**, **friend**, and **protected**. No variable shall have a name that overlaps with a variable name from the C Standard Library (e.g., **errno**). No variable shall have a name that begins with an underscore. No variable name shall be **longer than 31** characters. No variable name shall be **shorter than 3** characters, including loop counters. Each variable's name shall be descriptive of its purpose. The names of any **global variables** shall begin with the letter '**g**'. For example, **g_zero_offset**. The names of any **pointer variables** shall begin with the letter '**p**'. For example, **p_led_reg**. The names of any **pointer-to-pointer** variables shall begin with the letters '**pp**'. For example, **pp_vector_table.** The names of all **integer variables containing Boolean information** (including 0 vs. non-zero) shall begin with the letter '**b**' and phrased as the question they answer. For example, **b_done_yet** or **b_is_buffer_full**.

Calling Convention:

Calling conventions are a scheme for how functions receive parameters from their caller and how they return a result. Adhering to calling conventions ensures that your functions won't step on each other's data when using the same registers. Calling conventions allow us to implement recursive functions and call functions which we cannot see the implementations of. Certain registers need to have their contents preserved by the caller if the caller wants to ensure that the values in those registers are saved across the function call. Other registers need to have their contents saved by the callee (the function being called) before using them. When arguments are pushed onto the stack, eventually they must be popped back off again. Whichever function, the caller or the callee, is responsible for cleaning the stack must reset the stack pointer to eliminate the passed arguments.

C Calling Conventions

Keyword	Stack cleanup	Parameter passing
__cdecl	Caller	Pushes parameters on the stack, in reverse order (right to left)
__clrcall	n/a	Load parameters onto CLR expression stack in order (left to right).
__stdcall	Callee	Pushes parameters on the stack, in reverse order (right to left)
__fastcall	Callee	Stored in registers, then pushed on stack
__thiscall	Callee	Pushed on stack; this pointer stored in ECX
__vectorcall	Callee	Stored in registers, then pushed on stack in reverse order (right to left)

__cdecl

The default calling convention for C and C++ programs. Pushes parameters on the stack, in reverse order (right to left). Because the stack is cleaned up by the caller, it can do vararg functions. The **__cdecl** calling convention creates larger executables than __stdcall, because it requires each function call to include stack cleanup code. Consider the following code: **test_calling_convention.c**

What do you think the output of program will be?

~~Output: 1 2 3~~

Output: 3 3 1

As Calling convention is **Right to Left:**

- X value is assigned to x++ (Post Increment), arg value = 1
- x++ increments the value of x, new value of x = 2
- ++x due to pre increment new value of x = 3, arg value = 3
- Current X value = 3, arg value = 3

```c
#include <stdio.h>

int main()
{
    int x = 1;
    printf("%d\t%d\t%d\r\n", x , ++x, x++);
    return 0;
}
```

```
akshay@akshayv:Chapter10_Coding_Practices$ gcc test_calling_convention.c -o test_calling_convention
akshay@akshayv:Chapter10_Coding_Practices$ ./test_calling_convention
3       3       1
akshay@akshayv:Chapter10_Coding_Practices$
```

Comment

A **comment** is an explanation or description of the source code of the program. It helps a developer explain logic of the code and improves program readability. At run-time, a comment is ignored by the compiler.

Types of Comments: There are two ways to add comments in C:

- **//** - Single Line Comment
- **/*...*/** - Multi-line Comment

Advantages of commenting on a program:

- Using comments in a program, the program seems easy to read and maintain.
- By using comments to explain the program's working and logic made the program universally accepted as everyone finds it easy to use and understand.
- Using comments not only others but you can also understand your code after ages.

Comment Rules

Acceptable Formats:

Single-line comments in the C++ style (i.e., preceded by //) are a useful and acceptable alternative to traditional C style comments (i.e., /* … */). Comments shall never contain the preprocessor tokens /*, //, or \. Code shall never be commented out, even temporarily. To temporarily disable a block of code, use the preprocessor's conditional compilation feature (e.g., #if 0 … #endif). Any line or block of code that exists specifically to increase the level of debug output information shall be surrounded by :

#ifndef NDEBUG …

#endif.

Locations and Content

All comments shall be written in clear and complete sentences, with proper spelling and grammar and appropriate punctuation. The most useful comments generally precede a block of code that performs one step of a larger algorithm. A blank line shall follow each such code block. The comments in front of the block should be at the same indentation level. Avoid explaining the obvious. Assume the reader knows the C programming language. For example, end-of-line comments should only be used where the meaning of that one line of code may be unclear from the variable and function names and operations alone, but where a short comment makes it clear. Specifically, avoid writing unhelpful and redundant comments,

e.g., "numero <<= 2;

// Shift numero left 2 bits.".

The number and length of individual comment blocks shall be proportional to the complexity of the code they describe. All assumptions shall be spelled out in comments. Each module and function shall be commented in a manner suitable for automatic documentation generation, e.g., via **Doxygen**.

Conditional Compilation

Conditional compilation provides a way of including or omitting selected lines of source code depending on the values of literals specified by the directives. In this way, you can create multiple variants of the same program without the need to maintain separate source streams. There are different ways we can use Conditional compilation directives.

#if <condition that can be evaluated by the preprocessor >

…….

#endif

```c
#include <stdio.h>

#define DEBUG 0

int main()
{
#if DEBUG
    printf("This is not printed\r\n");
#endif
    printf("This is printed\r\n");
    return 0;
}
```

```
akshay@akshayv:Chapter10_Coding_Practices$ ./contidional_compilation_Sample2
This is printed
akshay@akshayv:Chapter10_Coding_Practices$
```

```c
#include <stdio.h>

int main()
{
#if 0
    printf("thisisnotprinted\n");
#endif
    printf("Thisisprinted\n");
    return 0;
}
```

```
akshay@akshayv:Chapter10_Coding_Practices$ ./contidional_compilation_Sample1
This is printed
akshay@akshayv:Chapter10_Coding_Practices$
```

#else

#if

…

#elif

…

#else

…

#endif

- **#ifdef** means "if defined", and is terminated by an #endif.
- **#indef** means "if not defined".
- **#undef** removes a previously defined definition.

- **#pragma** directive is an implementation-defined directive which allows the various instructions to be given to the compiler.

```c
#include <stdio.h>

#define DEBUG 1

int main()
{
#if (DEBUG == 0)
    printf("This is Debug Level 0 Message\r\n");
#elif (DEBUG == 1)
    printf("This is Debug Level 1 Message\r\n");
#elif (DEBUG == 2)
    printf("This is Debug Level 2 Message\r\n");
#endif

    return 0;
}
```

```
akshay@akshayv:Chapter10_Coding_Practices$ ./contidional_compilation_Sample3
This is Debug Level 1 Message
akshay@akshayv:Chapter10_Coding_Practices$
```

There are several reasons to use conditional compilation in your program:

- Code shall never be commented out, even temporarily.
- To temporarily disable a block of code, use the preprocessor's conditional compilation feature (e.g., #if 0 … #endif).
- Your program may require different code depending on which device or architecture you use.
- In some cases, library routines may exist in one configuration that do not exist in another.
- Conditional compilation allows you to handle such a situation by substituting alternate functions with the unavailable library routines.

Many complex functions require comprehensive test code to verify or validate I/O and proper operation.

- Intermediate values may be tested and output as well. After verification, you may wish to retain the test cases for future reference. You can include them in conditional blocks you control with a macro. For example:

```c
#include <stdio.h>

#define TEST_LEVEL 1

int main()
{
#if (TEST_LEVEL == 0)
    printf("This is TEST_LEVEL Level 0 Message\r\n");
#elif (TEST_LEVEL == 1)
    printf("This is TEST_LEVEL Level 1 Message\r\n");
#elif (TEST_LEVEL == 2)
    printf("This is TEST_LEVEL Level 2 Message\r\n");
#endif

    return 0;
}
```

UNIT 10.2 Secure and Safe Coding

Unit Objectives

At the end of this unit, you will be able to:

1. Avoid the pitfalls/limitation of various concepts in C while developing a code

10.2.1 Scope Rules

Scope = Lifetime

The area under which a variable or function is applicable. Variables and functions should be declared in the minimum scope from which all references to the identifier are still possible. When a larger scope than necessary is used, code becomes less readable, harder to maintain, and more likely to reference unintended variables.

Scope	Meaning
File Scope	Scope of a Identifier starts at the beginning of the file and ends at the end of the file. It refers to only those Identifiers that are declared outside of all functions. The Identifiers of File scope are visible all over the file Identifiers having file scope are global
Block Scope	Scope of a Identifier begins at opening of the block / '{' and ends at the end of the block / '}'. Identifiers with block scope are local to their block
Function Prototype Scope	Identifiers declared in function prototype are visible within the prototype
Function scope	Function scope begins at the opening of the function and ends with the closing of it. Function scope is applicable to labels only. A label declared is used as a target to goto statement and both goto and label statement must be in same function

Minimize the scope of variables and functions

The function counter() increments the global variable count and then returns immediately if this variable exceeds a maximum value. Assuming that the variable count is only accessed from this function, this example does not define count within the minimum possible scope. The variable count is declared within the scope of the counter() function as a static variable. The static modifier, when applied to a local variable (one inside of a function), modifies the lifetime (duration) of the variable so that it persists for as long as the program does and does not disappear between invocations of the function.

```
unsigned int count = 0;

void counter() {

 if (count++ >
MAX_COUNT) return;

 /* ... */
```

1

```
void counter() {

 static unsigned int count = 0;

 if (count++ > MAX_COUNT) return;

 /* ... */

}
```

2

Minimize the scope of variables and functions

The function f() is called only from within the function g(), which is defined in the same compilation unit. By default, function declarations are *extern*, meaning that these functions are placed in the global symbol table and are available from other compilation units.

The function f() is declared with internal linkage. This practice limits the scope of the function declaration to the current compilation unit and prevents the function from being included in the external symbol table. It also limits cluttering in the global name space and prevents the function from being accidentally or intentionally invoked from another compilation unit.

```
int f(int i) {
 /* Function definition
*/
}
int g(int i) {
int j = f(i);
 /* ... */
}
```

1

```
static int f(int i) {
/* Function definition */
}
int g(int i) {
int j = f(i);
/* ... */
}
```

2

10.2.2 Handling Array

Do not form or use out-of-bounds pointers or array subscripts

The function f() attempts to validate the index before using it as an offset to the statically allocated table of integers. However, the function fails to reject **negative index values**. When index is less than zero, the behavior of the addition expression in the return statement of the function is **undefined behavior**. One compliant solution is to detect and reject invalid values of index if using them in pointer arithmetic would result in an invalid pointer. Use an **unsigned** type to avoid having to check for negative values while still rejecting out-of-bounds positive values of index.

```
enum {TABLE SIZE = 100 };
static int table[ TABLE SIZE ];
 int *f (int index )
{
   if (index < TABLESIZE )
   {
      return table + index;
   }
   return NULL;
}
```

1

```
enum { TABLESIZE = 100 };
static int table[ TABLESIZE ];
 int *f ( int index )
 {
   if (index >= 0 && index < TABLESIZE )
   {
      return table + index;
   }
   return NULL;
}
```

2

```
enum { TABLESIZE = 100 };
static int table[ TABLESIZE ];
 int *f ( int index )
{
   if (index >= 0 && index < TABLESIZE )
   {
      return table + index;
   }
   return NULL;
}
```

3

Apparently Accessible Out-of-Range Index

Code example declares matrix to consist of 7 rows and 5 columns in row-major order. The function init_matrix iterates over all 35 elements in an attempt to initialize each to the value given by the function argument x. However, because multidimensional arrays are declared in C in row-major order. The function iterates over the elements in column-major order, and when the value of j reaches the value COLS during the first iteration of the outer loop. The function attempts to access element matrix [0][5]. Because the type of matrix is int [7][5], the j subscript is out of range, and the access has undefined behavior. Solution avoids using out-of-range indices by initializing matrix elements in the same row-major order as multidimensional objects are declared in C.

```
#include <stddef.h>
#define COLS 5
#define ROWS 7
static int matrix[ROWS][COLS];
void init_matrix(int x){
   for (size_t i = 0; i < COLS; i++) {
      for (size_t j = 0; j < ROWS; j++){
         matrix[ i ] [ j ] = x;
      }
   }
}
```

1

```
#include <stddef.h>
#define COLS 5
#define ROWS 7
static int matrix[ROWS][COLS];
void init_matrix(int x){
   for (size_t i = 0; i < ROWS; i++) {
      for (size_t j = 0; j < COLS; j++){
         matrix[ i ] [ j ] = x;
      }
   }
}
```

2

Null Pointer Arithmetic

This code allocates a block of memory and initializes it with some data. The data does not belong at the beginning of the block, which is left uninitialized. Instead, it is placed offset bytes within the block. The function ensures that the data fits within the allocated block.

This function fails to check if the allocation succeeds. If the allocation fails, then malloc() returns a null pointer. The null pointer is added to offset and passed as the destination argument to memcpy(). Because a null pointer does not point to a valid object, the result of the pointer arithmetic is undefined behaviour. Solution ensures that the call to malloc() succeeds.

```
#include <string.h>
#include <stdlib.h>
char *init_block(size_t block_size, size_t offset, char *data, size_t data_size)
{
   char *buffer = malloc(block_size);
   if (data_size > block_size || block_size - data_size < offset)
   {
      /* Data won't fit in buffer, handle error */
   }
   memcpy(buffer + offset, data, data_size);
   return buffer;
}
```

1

```
#include <string.h>
#include <stdlib.h>
char *init_block(size_t block_size, size_t offset, char *data, size_t data_size)
{
   char *buffer = malloc(block_size);
   if (NULL == buffer)
   {
      /* Handle Error if malloc fails */
   }
   if (data_size > block_size || block_size - data_size < offset)
   {
      /* Data won't fit in buffer, handle error */
   }
   memcpy(buffer + offset, data, data_size);
   return buffer;
}
```

2

10.2.3 Control Statement Expressions

Control statement expressions

Assignment operators shall not be used in expressions that yield a Boolean value

No assignments are permitted in any expression which is considered to have a Boolean value. This precludes the use of both simple and compound assignment operators in the operands of a Boolean-valued expression. However, it does not preclude assigning a Boolean value to a variable. If assignments are required in the operands of a Boolean-valued expression then they must be performed separately outside of those operands. This helps to avoid getting “=” and “==” confused,

and assists the static detection of mistakes. 1,2 Does not follow the rule. We can use 3 to satisfy the rule.

```
if ( ( x = y ) != 0 ) /* Boolean by context */
{ `
   foo();
}
```

1

```
if ( x = y )
{
   foo();
}`
```

2

```
x = y;
if ( x != 0 )
{
   foo();
}
```

3

Control statement expressions

Tests of a value against zero should be made explicit, unless the operand is effectively Boolean.

Where a data value is to be tested against zero then the test should be made explicit. The exception to this rule is when data represents a Boolean value, even though in C this will in practice be an integer. This rule is in the interests of clarity, and makes clear the distinction between integers and logical values.

- Not compliant, unless y is effectively Boolean data e.g a flag
- Correct way of testing x is non-zero

```
if ( y )
```

1

```
if ( x != 0 )
```

2

Floating-point expressions shall not be tested for equality or inequality.

The inherent nature of floating-point types is such that comparisons of equality will often not evaluate to true even when they are expected to. In addition, the behaviour of such a comparison cannot be predicted before execution, and may well vary from one implementation to another. For example, the result of the test in the following code is unpredictable.

```
float32_t x, y;
if ( x == y ) { .... }
```

```
float32_t x, y;
if ( ( x <= y ) && ( x >= y ) )
{ .... }
```

```
float32_t x, y;
if ( x == 0.0f ) { .... }
```

Numeric variables being used within a for loop for iteration counting shall not be modified in the body of the loop.

Loop counters shall not be modified in the body of the loop. However other loop control variables representing logical values may be modified in the loop. For example, a flag to indicate that something has been completed, which is then tested in the for statement.

```
flag = 1;
for ( i = 0; (i < 5) && (flag == 1); i++ )
{
   /* ... */
   flag = 0;  /* Compliant - allows early termination of loop */
   i = i + 3; /* Not compliant - altering the loop counter */
}
```

The controlling expression of a for statement shall not contain any objects of floating type.

The controlling expression may include a loop counter, whose value is tested to determine termination of the loop. Floating-point variables shall not be used for this purpose. Rounding and truncation errors can be propagated through the iterations of the loop, causing significant inaccuracies in the loop variable, and possibly giving unexpected results when the test is performed.

10.2.4 Functions

Functions Definitions:

Functions shall not call themselves, either directly or indirectly:

This means that recursive function calls cannot be used in safety-related systems. Recursion carries with it the danger of exceeding available stack space, which can be a serious error. Unless recursion is very tightly controlled, it is not possible to determine before execution what the worst-case stack usage could be.

Identifiers shall be given for all of the parameters in a function prototype declaration:

Names shall be given for all the parameters in the function declaration for reasons of compatibility, clarity and maintainability.

- The identifiers used in the declaration and definition of a function shall be identical.
- Functions with no parameters shall be declared and defined with the parameter list void.
- All exit paths from a function with non-void return type shall have an explicit return statement with an expression.

This expression gives the value that the function returns. The absence of a return with an expression leads to undefined behaviour (and the compiler may not give an error)

If a function returns error information, then that error information shall be tested:

A function (whether it is part of the standard library, a third-party library or a user defined function) may provide some means of indicating the occurrence of an error. The calling program shall check for the indication of an error as soon as the function returns.

Functions

Functions shall not be defined with a variable number of arguments:

There are a lot of potential problems with this feature. Users shall not write additional functions that use a variable number of arguments. This precludes the use of use of stdarg.h, va_arg, va_start and va_end.

```
void MyFunc ( char_t * pBuf, ... )
{
  // ...
}
```

Function should be declared explicitly:

The use of function prototype enables the compiler to check the integrity of function definitions and calls.

```
void example() {
  fun();
}
void fun() {
}
```

```
void fun(void);
void example(void) {
  fun();
}
void fun(void) {
}
```

Functions Parameters should not be reassigned:

```
int globalVar = 0;
void function (int a) {
  a = globVar;
  ...
}
```

```
int globalVar = 0;
void function (int a) {
  int b = globVar;
  ...
}
```

Function should not contain too many returns:

Having too many returns from function increases the function complexity as flow of execution is broken each time return statement is encountered

```
int fun() {
  if (condition1) {
    return 1;
  } else {
    if (condition2) {
      return 0;
    } else {
      return 1;
    }
  }
  return 0;
}
```

10.2.5 Debug Port

If left unlocked, the debug port constitutes a significant security vulnerability. Thus, the security best practice is to lock or disable debug access before a product is released to production. If you need to debug in production, there are a few things to ensure when doing so;

- Debugging does not have an extensive performance hit.
- Debugging does not block people from using your application.
- Secure data is not exposed to the outside world.
- You are getting enough debug information to find and fix the issue as soon as possible.

Specific debugging methods, such as a remote debugger in your IDE or dumping error information to the user, are not considered safe. If your product is Linux based system always disable the serial port and ssh login access usually used in development phase to the device in final image.

10.2.6 Buffer Overflow

Buffer overflow is a common security vulnerability in C programming. A buffer overflow occurs when a program allows input to write data beyond allocated memory. The attacker can gain control over an entire application or crash a program by exploitation via buffer overflow.

How Buffer Overflow Invites Hackers

Hackers can use buffer overflow errors to cause a program to crash, corrupt the data, or simply steal information. When a program runs, it uses an area of memory referred to as the 'stack'. Variables within the scope of the currently executing function will be stored on the stack. The address of the function call will also be stored to allow return statements to return to the correct location. When the function returns to the calling function, the program execution continues from where it left off. So, if the return address on the stack is modified to point to some alternative malicious instructions, then those instructions will be executed when the function returns.

```c
/** Example of Buffer Overflow  **/
#include <stdio.h>
#include <string.h>

int main(int argc, char const *argv[])
{
    char buffer1[5] = "VXYZ";
    char buffer2[5] = "PQRS";
    // The argument is copied into buffer 2 without checking its size.
    // This flaw introduces a buffer overflow vulnerability.
    strcpy(buffer2, argv[1]);

    printf("buffer1: %s, bufffer2: %s\n", buffer1, buffer2);
    return 0;
}
```

```
TERMINAL

akshay@akshayv:Chapter10_Coding_Practices$ ./BufferOverFlow_examples
Segmentation fault (core dumped)
akshay@akshayv:Chapter10_Coding_Practices$ ./BufferOverFlow_examples  123
buffer1: VXYZ, bufffer2: 123
akshay@akshayv:Chapter10_Coding_Practices$ ./BufferOverFlow_examples  1234567890
buffer1: VXYZ, buffer2: 1234567890
*** stack smashing detected ***: terminated
Aborted (core dumped)
akshay@akshayv:Chapter10_Coding_Practices$
```

10.2.7 Regular Cleanup

If any dynamic allocation is in the program, the memory should be freed if no longer needed by the programmer.

Use of Garbage Collector:

In computer science, garbage collection (GC) is a form of automatic memory management. The garbage collector attempts to reclaim memory which was allocated by the program, but is no longer referenced; such memory is called garbage. Garbage Collection (GC) is a mechanism that provides automatic memory reclamation for unused memory blocks. The GC engine takes care of recognizing that a particular block of allocated memory (heap) is not used anymore and puts it back into the free memory area. The Boehm-Demers-Weiser (BDW) GC library, a popular package that allows C and C++ programmers to include automatic memory management into their programs. This is a garbage collecting storage allocator that is intended to be used as a plug-in replacement for C's malloc. BWGC The library is freely available on GitHub.

UNIT 10.3 Develop Optimal Code

Unit Objectives

At the end of this unit, you will be able to:

1. Write an optimal code in terms of memory and performance.

10.3.1 Macros vs Functions

A macro is a name given to a block of C statements as a pre-processor directive. Being a pre-processor, the block of code is communicated to the compiler before entering into the actual coding (main () function). A macro is defined with the pre-processor directive. Macros are pre-processed, which means that all the macros would be processed before your program compiles. However, functions are not preprocessed but compiled. In macros, no type checking is done, so it may occur some problems for different types of inputs. In the case of functions, this is not done. For macros if the inputs are not properly maintained, then it may generate some invalid results. Difficult to debug as they cause simple replacement. Macro don't have namespace, so a macro in one section of code can affect other section.

```c
#include <stdio.h>
// Macro defination to calculate square of argument:
#define SQUARE(x) (x * x)
// Function to calculate the square of argument:
int sqr(int x)
{
    return x * x;
}
int main()
{
    printf("Use of Func : sqr()    - The value of sqr(8+2)    : %d\r\n", sqr(8 + 2));
    printf("Use of Macro: SQUARE() - The value of SQUARE(8+2): %d\r\n", SQUARE(8 + 2));
    return 0;
}
```

```
akshay@akshayv:Chapter10_Coding_Practices$ gcc macro_func_demo.c -o macro_func_demo
akshay@akshayv:Chapter10_Coding_Practices$ ./macro_func_demo
Use of Func : sqr()    - The value of sqr(8+2)    : 100
Use of Macro: SQUARE() - The value of SQUARE(8+2): 26
akshay@akshayv:Chapter10_Coding_Practices$
```

Consider example: **macro_func_demo.c**

The function and macro, we want both will do the same task, but here we can see that the output are not same. The main reason is when we are passing 8 + 2 as function argument, it converts into 10, then calculate 10 * 10 = 100.

For macro it is doing 8 + 2 * 8 + 2 = 8 + 16 + 2 = 26.

If you need to choose between a function and a macro implementation of a library routine, consider the following trade-offs:

Speed versus size

The main benefit of using macros is faster execution time. During preprocessing, a macro is expanded (replaced by its definition) inline each time it's used. A function definition occurs only once regardless of how many times it's called. Macros may increase code size but don't have the overhead associated with function calls.

Function evaluation

A function evaluates to an address; a macro doesn't. Thus you can't use a macro name in contexts requiring a pointer. For instance, you can declare a pointer to a function, but not a pointer to a macro.

Type-checking

When you declare a function, the compiler can check the argument types. Because you can't declare a macro, the compiler can't check macro argument types; although it can check the number of arguments you pass to a macro.

10.3.2 Inline Functions

When an inline function is called, a copy of the C/C++ source code for the function is inserted at the point of the call. This is known as inline function expansion, commonly called function inlining or just inlining function expansion can speed up execution by eliminating function call overhead. This is particularly beneficial for very small functions that are called frequently. Function in lining involves a tradeoff between execution speed and code size, because the code is duplicated at each function call site. Large functions that are called in many places are poor candidates for in lining. Excessive in lining can make the compiler dramatically slower and degrade the performance of generated code.

Inlining functions might make it larger:

For example, if a system has 100 inline functions each of which expands to 100 bytes of executable code and is called in 100 places, that's an increase of 1MB. Is that 1MB going to cause problems? Who knows, but it is possible that that last 1MB could cause the system to "thrash," and that could slow things down.

Inline functions might make it smaller:

The compiler often generates more code to push/pop registers/parameters than it would by inline-expanding the function's body. This happens with very small functions, and it also happens with large functions when the optimizer is able to remove a lot of redundant code through procedural integration — that is, when the optimizer is able to make the large function small.

10.3.3 Recursion

An algorithmic technique where a function, in order to accomplish a task, calls itself with some part of the task. The power of recursion evidently lies in the possibility of defining an infinite set of objects by a finite statement. In the same manner, an infinite number of computations can be described by a finite recursive program, even if this program contains no explicit repetitions. Recursion carries with it the danger of exceeding available stack space, which can lead to a serious failure. Unless recursion is very tightly controlled, and it is not possible to determine before execution what the worst-case stack usage could be. That’s why recursive function calls cannot be used in safety-related systems. Recursive calls should be replaced with loops wherever possible. The following example demonstrates how this can be applied to the code: **test_factorial.c**

```c
#include <stdint.h>
#include <stdio.h>
// Function to calculate the factorial
// Uses loop:
uint32_t factorial(uint32_t n)
{
  uint32_t result = 1;
  for (; n > 1; --n)
  {
    result *= n;
  }

  return result;
}

// Function to calculate the factorial
// Uses recursion:
uint32_t factorialRecursive(uint32_t n)
{
  return n > 1 ? n * factorialRecursive(n - 1) : 1;
}
```

```
TERMINAL

akshay@akshayv:Chapter10_Coding_Practices$ gcc test_factorial.c -o test_factorial
akshay@akshayv:Chapter10_Coding_Practices$ ./test_factorial
Factorial using: factorial(5)          :  120
Factorial using: factorialRecursive(5) :  120
akshay@akshayv:Chapter10_Coding_Practices$
```

10.3.4 Memory Overlay

In a general computing sense, overlaying means "the process of transferring a block of program code or other data into main memory, replacing what is already stored". Overlaying is a programming method that allows programs to be larger than the computer's main memory. An embedded system would normally use overlays because of the limitation of physical memory, which is internal memory for a system-on-chip, and the lack of virtual memory facilities. The concept of overlays is that whenever a process is running it will not use the complete program at the same time, it will use only some part of it. Then overlays concept says that whatever part you required, you load it and once the part is done, then you just unload it, means just pull it back and get the new part you required and run it.

In memory management, overlays work in the following steps, such as:

The programmer divided the program into many logical sections. A small portion of the program had to remain in memory at all times, but the remaining sections (or overlays) were loaded only when needed. The use of overlays allowed programmers to write programs much larger than physical memory, although memory usage depends on the programmer rather than the operating system.

Example of Overlays

The best example of overlays is assembler.

Consider the assembler has 2 passes, 2 pass means at any time it will be doing only one thing, either the 1st pass or the 2nd pass.

This means it will finish 1st pass first and then 2nd pass.

Let's assume that the available main memory size is 150KB and the total code size is 200KB.

- Pass 1.....70KB
- Pass 2.....80KB
- Symbol table.....30KB
- Common routine.....20KB

As the total code size is 200KB and the main memory size is 150KB, it is impossible to use 2 passes together. So, in this case, we should go with the overlays technique.

- According to the overlay concept, only one pass will be used, and both the passes always need a symbol table and common routine.
- If the overlays driver is 10KB, what minimum partition size is required?
- For pass 1 total memory needed is = (70KB + 30KB + 20KB + 10KB) = 130KB.
- For pass 2 total memory needed is = (80KB + 30KB + 20KB + 10KB) = 140KB.
- So if we have a minimum 140KB size partition, we can run this code very easily.

So overlay is a technique to run a program that is bigger than the size of the physical memory by keeping only those instructions and data that are needed at any given time. Divide the program into modules in such a way that not all modules need to be in the memory at the same time.

Advantages:

- Reduce memory requirement
- Reduce time requirement

Disadvantage:

- Overlap map must be specified by programmer
- Programmer must know memory requirement
- Overlapped module must be completely disjoint
- Programming design of overlays structure is complex and not possible in all cases

10.3.5 Structure Padding

Structure members are assigned to memory addresses in increasing order, with the first component starting at the beginning address of the structure name itself. Structure padding is a concept in C that adds the one or more empty bytes between the memory addresses to align the data in memory. The processor does not read 1 byte at a time. It reads 1 word at a time.

What does the 1 word mean?

If we have a **32-bit processor**, then the processor reads 4 bytes at a time **1 word = 4 bytes.**

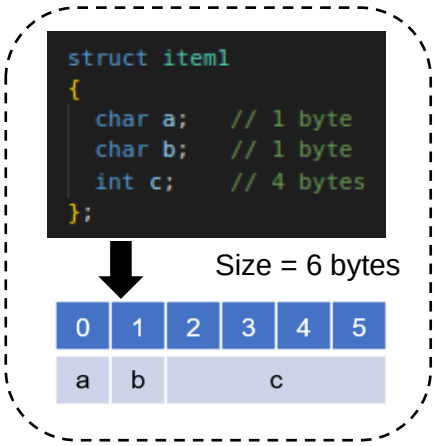
If we have a **64-bit processor**, then the processor reads 8 bytes at a time, **1 word = 8 bytes.**

- As we know that structure occupies the contiguous block of memory as shown in diagram.
- Consider the **32-bit architecture:**

The problem is that in one CPU cycle we can access:

- one byte of **char a**,
- one byte of **char b**,
- bytes of **int c**.

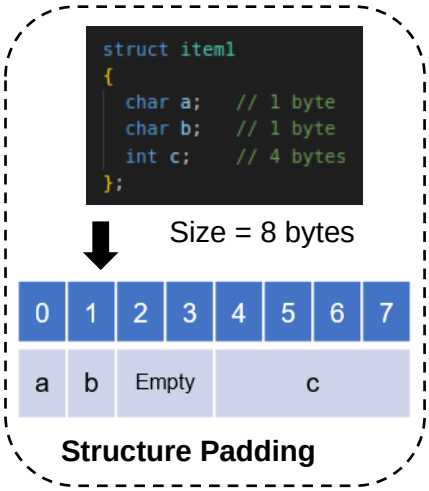
We will not face any problem while accessing the **char a** and **char b** as both the variables can be accessed in **one CPU Cycle**. We will face the problem when we access the **int c** variable as **2 CPU cycles** are required to access the value of the 'c' variable.



Suppose we only want to access the variable 'c', which requires two cycles. The variable 'c' is of 4 bytes, so it can be accessed in one cycle also, but in this scenario, it is utilizing 2 cycles. This is an unnecessary wastage of CPU cycles. Due to this reason, the structure padding concept was introduced to save the number of CPU cycles.

How is structure padding done?

To achieve the structure padding, an empty row is created on the left, as shown in the diagram. The two bytes which are occupied by the 'c' variable on the left are shifted to the right. So, all the four bytes of 'c' variable are on the right. Now, the 'c' variable can be accessed in a single CPU cycle. After structure padding, the total memory occupied by the structure is 8 bytes (1 byte+1 byte+2 bytes+4 bytes). Although the memory is wasted in this case, the variable can be accessed within a single cycle.



The structural padding is an in-built process that is automatically done by the compiler. Sometimes it required to avoid the structure padding in C as it makes the size of the structure greater than the size of the structure members.

Structure Packing:

Packing, prevents compiler from doing padding means remove the unallocated space allocated by structure.

We can avoid the structure padding in C in two ways:
Using #pragma pack (1) directive

- Using __attribute__((__packed__))

That will allow us to save some memory.

```c
// To avoid the structure padding.
// Use the #pragma pack(1) directive
#pragma pack(1)
struct item3
{
    char a;  // 1 byte
    char b;  // 1 byte
    int  c;  // 4 byte
};

// By using attribute
struct item4
{
    char a;  // 1 byte
    char b;  // 1 byte
    int  c;  // 4 bytes
}__attribute__((packed));;
```

Example Program: Demo for strcut padding: test_structure_padding.c

```
akshay@akshayv:Chapter4_Advanced_Data_Types$ gcc test_structure_padding.c -o test_structure_padding
akshay@akshayv:Chapter4_Advanced_Data_Types$ ./test_structure_padding
The size of the item1 structure = 8
The size of the item2 structure = 12
The size of the item3 structure = 6
The size of the item4 structure = 6
akshay@akshayv:Chapter4_Advanced_Data_Types$
```

Structure Good Practice:

If you don't want to use the structure packing and still want to reduce the memory wastage, then you need to correctly order the structure. We need to declare members in decreasing or increasing order of size. We have given two structures called **temp1** (members are not ordered) and **temp2** (members are ordered). We have saved 4 bytes without using the structure packing and just changing the order of the members. But still, the last two bytes are padded and we cannot avoid that. We can use this method also to reduce memory wastage.

```c
#include <stdio.h>

// Members are not ordered
typedef struct
{
  char a;   // 1 byte
  int  b;   // 4 bytes
  char c;   // 1 byte
} temp1;

//Members are ordered
typedef struct
{
  int  b;   // 4 bytes
  char a;   // 1 byte
  char c;   // 1 byte
} temp2;

int main()
{
  printf("sizeof(structure temp1) = %lu\r\n", sizeof(temp1));
  printf("sizeof(structure temp2) = %lu\r\n", sizeof(temp2));
  return 0;
}
```

```
akshay@akshayv:Chapter10_Coding_Practices$ gcc Structure_better_practice.c -o Structure_better_practice
akshay@akshayv:Chapter10_Coding_Practices$ ./Structure_better_practice
sizeof(structure temp1) = 12
sizeof(structure temp2) = 8
akshay@akshayv:Chapter10_Coding_Practices$
```

10.3.6 Optimize for Space

Avoid use of inline function it is used at many places.

Avoid Standard Library Functions:

One of the best things you can do to reduce the size of your program is to avoid using large standard library routines. Many of the largest are expensive only because they try to handle all possible cases. It might be possible to implement a subset of the functionality yourself with significantly less code.

For reducing the size, we can rely on compiler optimization for size flag -Os optimize for code size.

This uses O2 as the baseline, but disables some optimizations. For example, it will not inline[3] code if that leads to a size increase. The first time you enable the compiler's optimization feature your previously working program will suddenly fail. Perhaps the most notorious of the automatic optimizations is "dead code elimination." This optimization eliminates code that the compiler believes to be either redundant or irrelevant. For example, adding zero to a variable requires no run-time calculation whatsoever.

Never make the mistake of assuming that the optimized program will behave the same as the unoptimized one.

Optimize for Performance

Switch Statement

Compilers translate switch statements in different ways. If case labels are small contiguous integer values, then it creates a jump table. This is very fast and doesn't depend on the number of case labels also. If case labels are longer and not contiguous then it creates comparison tree i.e. if…else statements. So, in this case, we should keep the most frequent label first and the least frequent label should be at last.

Fast Mathematics

Multiplication and division by power of 2 : Use left shift(<<) for multiplication and right shift(>>) for division.

The bit operations will be much faster than multiplication and division operations.

Multiply by 6 : a= a<<1 + a<<2;

Multiply by 7 : a= a<<3 - a;

Divide by 8 : a= a>>3; // division by power of 2

Simplifying Expressions: Sometimes we can reduce some operations by simplifying expressions.

a*b + a*b*c + a*b*c*d ---> (a*b)*(1 + c*(1 + d))

L.H.S can be Simplified to R.H.S

L.H.S : 6 multiplications and 2 additions

R.H.S : 3 multiplications and 2 additions

Unroll small loops:

Most of the times Compiler does this automatically, but it is a good habit of writing optimized codes. Matrix updations using this is very advantageous.

```
#include <stdio.h>
int main(void)
{
   int fact[5];
   fact[0] = 1;
   // Overhead of managing a counter
   // just for 4 iterations
   // is not a good idea
   for (int i = 1; i < 5; ++i) {
      fact[i] = fact[i - 1] * i;
   }
   return 0;
}
```

```
#include <stdio.h>
int main(void)
{
   int fact[5] = { 1, 1, 2, 6, 24 };
   // Here the same work is done
   // without counter overhead
   return 0;
}
```

Avoid calculations in loop:

We should avoid any calculation which is more or less constant in value. Inner loops should have minimum possible calculations.

```c
#include <stdio.h>
int main(void)
{
   int arr[1000];
   int a = 1, b = 5, c = 25, d = 7;
   // Calculating a constant
expression
   // for each iteration is not good.
   for (int i = 0; i < 1000; ++i) {
      arr[i] = (((c % d) * a / b) % d) * i;
   }
   return 0;
}
```

```c
#include <stdio.h>
int main(void)
{
   int arr[1000];
   int a = 1, b = 5, c = 25, d = 7;
   // pre calculating the constant expression
   int temp = (((c % d) * a / b) % d);
   for (int i = 0; i < 1000; ++i) {
      arr[i] = temp * i;
   }
   return 0;
}
```