

# **NXKR\_YoungSikYang's Home**

NXKR\_YoungSikYang

Exported on 03/16/2024

# Table of Contents

About Me (NXKR_YoungSikYang).....	13
Navigate space .....	14
Linux kernel .....	15
1. Linux Device driver.....	15
1. Types of devices in Linux .....	15
1.1 Character Devices.....	15
1.2 Block Devices.....	16
1.3 Network Devices .....	16
2. Misc Driver .....	16
3. Module Driver.....	16
3.1 Explanation.....	16
3.2 Diagram showing how LKMs are loaded .....	17
4. Adding a dummy module driver to the kernel .....	17
4.1 Source code .....	17
4.2 Build the module driver.....	21
4.3 Loading the module driver .....	23
4.4 Performing operations of the loaded module driver .....	25
4.4.1 Source code .....	25
4.4.2 Result.....	25
Reference .....	25
2. Device-Tree.....	26
Device tree.....	26
1. Understanding the device tree .....	26
2. How the device tree is loaded by Bootloader.....	26
3. Platform device - Platform driver.....	26
4. How to view the device tree in linux .....	29
Memory set-up .....	30

1. Disable the default DRAM setup .....	30
2. Memory size.....	31
3. Reserved memory .....	32
4. CMA(Contiguous Memory Allocator) .....	33
Reference .....	34
3. Driver sequence .....	34
1. Kernel initialization .....	34
1.1 Kernel initialization level.....	34
1.2 Kernel initialization process.....	35
1.2.1 Flow of function calls .....	35
1.2.2 Registration of initcalls .....	36
1.3 Diagram showing how modules are loaded in the kernel initialization.....	38
2. Module driver .....	39
2.1 Registration and loading of a module driver.....	39
2.2 Registering a module driver in 'postcore' .....	40
Reference .....	41
4. Interrupt .....	41
1. Polling.....	41
2. About interrupt .....	42
2.1 Types of the interrupt .....	42
2.2 Steps of an interrupt.....	42
2.3. ISR(Interrupt Service Routine).....	43
3. Interrupt practice .....	45
3.1 Directly in the top-half .....	46
3.2 Using softIRQ .....	47
3.3 Using tasklet .....	50
3.4 Using workqueue .....	52
3.5 Through threaded IRQ .....	56
Reference .....	57

5. Lock .....	58
1. Terminology .....	58
2. Lock mechanisms.....	59
3. How to implement a lock .....	60
3. Lock in the interrupt context .....	60
4. Practice.....	61
4.1 Chattering and debouncing.....	61
4.2 Code flow .....	62
4.3 Example of race condition without a lock .....	63
4.4 Example of race condition with a lock .....	65
6. Memory allocation .....	67
1. Memory allocation APIs .....	67
1.1 kmalloc() .....	67
1.2 vmalloc() .....	70
1.3 dma-alloc() .....	70
1.4 Other APIs .....	70
2. SLUB .....	71
Features and Advantages.....	71
How It Works .....	71
3. Practice.....	71
3.1 kmalloc() .....	74
4.2 vmalloc() .....	74
4.3 dma_alloc_coherent().....	75
7. Timer.....	76
1. Jiffies .....	76
1.1 Uses .....	76
1.2 Limitation .....	76
1.3 Resolution change .....	76
2. Hrtimer.....	77

2.2 Uses .....	77
3. getnstimeofday .....	77
3.1 Overview .....	78
4. Practice.....	78
4.1 Jiffies.....	78
4.2 Getnstimeofday .....	80
4.3 Hrtimer.....	80
Reference .....	82
Memo.....	83
U-boot .....	84
1. Device Driver .....	84
1 .....	84
Driver Model .....	84
Device Tree.....	85
FDT.....	86
menuconfig .....	89
2. U_BOOT_CMD .....	94
How to add and delete a custom command in u-boot .....	94
LD script.....	97
3. GPIO.....	99
Check which GPIO the LED is connected to.....	99
Device tree.....	99
Data sheet .....	99
GPIO control .....	100
2. KernelBoot.....	103
Difference between kernel images .....	103
Image .....	103
zImage .....	103
ulimage.....	103

How to compile the linux kernel to make Image and zImage .....	104
Install dependencies .....	104
Build (for ARM32) .....	104
Build (for ARM64) .....	104
How to make ulimage .....	105
Copy the compiled kernel image to wrap around the ulimage into uboot-2016.01/tools .....	105
Create ulimage .....	105
How to boot the created image (ulimage in this example) .....	105
Check the boot command with \$ printenv .....	105
Load the image into the RAM of the board .....	105
Run the ulimage in the target board .....	106
booti, bootz .....	107
How to upload data into the eMMC using fastboot .....	108
File system .....	108
partmap_emmc.txt .....	109
Upload .....	109
3. MMU .....	109
Enable cache .....	109
common/cmd_cache.c for the cache commands .....	109
Compile cmd_cache.c .....	110
Enable the cache commands .....	110
Enable dcache(disabled by default) .....	110
Check icache and dcache are enabled .....	110
Check the speed when the cache is on and off .....	111
icache(Instruction cache) .....	111
dcache(Data cache) .....	111
Cache in MMU page table .....	111
Cache in memory map .....	112

4. Device Driver assignment.....	114
INDEX.....	114
0. UPDATE HISTORY.....	114
1. Introduction .....	114
2. Implemented features .....	115
2.1 Feature list.....	115
2.2 Command manual .....	115
2.3 Flow diagram .....	115
3. Target GPIO pins.....	116
4. Source.....	117
4.1 Source code .....	117
4.2 Add the source code.....	120
4.3 Existing APIs used in the source code .....	121
5. Test .....	121
5.1 Test scenario .....	121
5.2 Test code.....	122
5.3 Run test .....	126
5.4 Test result .....	126
Reference .....	129
양영식 .....	130
ARM 양영식 .....	130
1. Introduction to the ARM Architecture.....	130
Introduction to the ARM Architecture.....	130
2. Programmers' model.....	133
Programmers' Model .....	133
3. Memory.....	143
Memory.....	143
BUS 양영식 .....	152
APB bus 양영식 .....	152

APB .....	152
Prior knowledge .....	153
AXI bus 양영식 .....	160
Introduction .....	160
Key features of AXI .....	160
Architecture .....	160
Channel Handshake .....	163
Addressing Options .....	165
Atomic Accesses .....	168
Response Signaling .....	169
Ordering Model .....	171
Data buses .....	172
Unaligned Transfer .....	174
Clock and Reset .....	175
Low-power Interface .....	176
Linux .....	180
12.19 과제 .....	180
Chapter 1 .....	180
Chapter 2 .....	180
12.20 과제 .....	182
12.21,22 과제 .....	182
12.26 chapter 06 .....	183
12.27 과제 .....	184
2 .....	184
3 .....	185
4 .....	185
5 .....	185
7 .....	185
8 .....	185

9 .....	185
10 .....	185
11 .....	186
18 .....	186
19 .....	186
27 .....	186
Yocto 양영식 .....	187
1st exercise .....	187
1: Create and execute the image of Arm64 Weston/wayland .....	187
2: .....	189
2nd exercise .....	191
1: .....	191
2: .....	195
Final exercise .....	196
Set up .....	196
Build image .....	198
Verify .....	198
Trouble-shooting .....	200
자료구조, 알고리즘 .....	201
1. Linked list 양영식 .....	201
Linked list란? .....	201
Array와 Linked list의 비교 .....	201
2. Doubly Linked List 양영식 .....	206
3. Circular doubly linked list 양영식 .....	211
4. ArrayStack 양영식 .....	216
Advantages .....	216
Problems of this code .....	216
5. LinkedListStack 양영식 .....	219
결과 .....	219

코드.....	219
6. chapter02 연습문제, 사칙연산 양영식 .....	223
연습문제 1 .....	223
연습문제 2 .....	223
연습문제 3 .....	224
사칙연산.....	224
7. Circular queue 양영식.....	234
8. LinkedQueue 양영식 .....	237
10. 트리(LCRS, Binary) 양영식 .....	240
11. Expression Tree 양영식.....	240
12. Union-Find .....	240
13. Sort .....	240
17. Priority queue.....	244
Priority queue .....	244
chapter 06 탐색.....	247
14. BinarySearch .....	247
15. BinarySearchTree .....	248
16. RedBlackTree .....	253
Chapter 8 해시테이블.....	264
18. Simple hash table .....	264
19. Chaining.....	268
20. Open addressing.....	273
컴퓨터구조 .....	279
01 컴퓨터 구조 시작하기 .....	279
컴퓨터 구조의 큰 그림 .....	279
02 데이터.....	281
0과 1로 숫자를 표현하는 방법 .....	281
How to represent decimals .....	283
0과 1로 문자를 표현하는 방법 .....	284

03 명령어.....	285
소스 코드와 명령어.....	285
Just In time Compile.....	286
명령어의 구조.....	286
04 CPU의 작동 원리.....	288
ALU와 제어장치 .....	288
명령어 사이클과 인터럽트 .....	290
05 CPU 성능 향상 기법 .....	292
빠른 CPU를 위한 설계 기법 .....	292
multi-core .....	292
thread.....	292
명령어 병렬 처리 기법 .....	292
Superscalar.....	293
Instruction set .....	293
06 메모리와 캐시 메모리 .....	294
RAM의 특징과 종류.....	294
메모리의 주소 공간.....	296
Cache memory .....	297
08 입출력장치 .....	298
장치 컨트롤러와 장치 드라이버 .....	298
다양한 입출력 방법.....	298
09 운영체제 시작하기, 12 Process synchronization .....	299
운영체제의 큰 그림.....	299
운영체제의 핵심 서비스.....	299
Process synchronization.....	299
동기화 기법 .....	300
13 deadlock.....	302
Deadlock.....	302
14 가상 메모리 .....	305

Contiguous memory allocation.....	305
Non-contiguous memory allocation .....	306

# About Me (NXKR\_YoungSikYang)

You may edit this page to include additional information about yourself.

E-mail: None

## Recently Updated

-  [1. Linux Device driver \(see page 15\)](#)  
2024.Mar.13 • updated by NXKR\_YoungSikYang • view change
-  [5. Lock \(see page 58\)](#)  
2024.Mar.13 • updated by NXKR\_YoungSikYang • view change
-  [Memo \(see page 83\)](#)  
2024.Mar.13 • updated by NXKR\_YoungSikYang • view change
-  [4. Device Driver assignement \(see page 114\)](#)  
2024.Mar.12 • updated by NXKR\_YoungSikYang • view change
-  [image2024-1-25\\_16-14-16.png](#)  
2024.Mar.12 • attached by NXKR\_YoungSikYang
-  [image2024-1-25\\_16-17-41.png](#)  
2024.Mar.12 • attached by NXKR\_YoungSikYang
-  [image2024-1-25\\_16-22-55.png](#)  
2024.Mar.12 • attached by NXKR\_YoungSikYang
-  [image2024-1-25\\_17-30-7.png](#)  
2024.Mar.12 • attached by NXKR\_YoungSikYang
-  [image2024-1-25\\_17-32-44.png](#)  
2024.Mar.12 • attached by NXKR\_YoungSikYang
-  [image2024-1-25\\_17-34-37.png](#)  
2024.Mar.12 • attached by NXKR\_YoungSikYang
-  [Final exercise \(see page 196\)](#)  
2024.Mar.12 • created by NXKR\_YoungSikYang
-  [image2024-1-24\\_15-23-49.png](#)  
2024.Mar.12 • attached by NXKR\_YoungSikYang
-  [image2024-1-24\\_15-24-13.png](#)  
2024.Mar.12 • attached by NXKR\_YoungSikYang
-  [image2024-1-24\\_15-51-22.png](#)  
2024.Mar.12 • attached by NXKR\_YoungSikYang
-  [image2024-1-24\\_16-1-46.png](#)  
2024.Mar.12 • attached by NXKR\_YoungSikYang

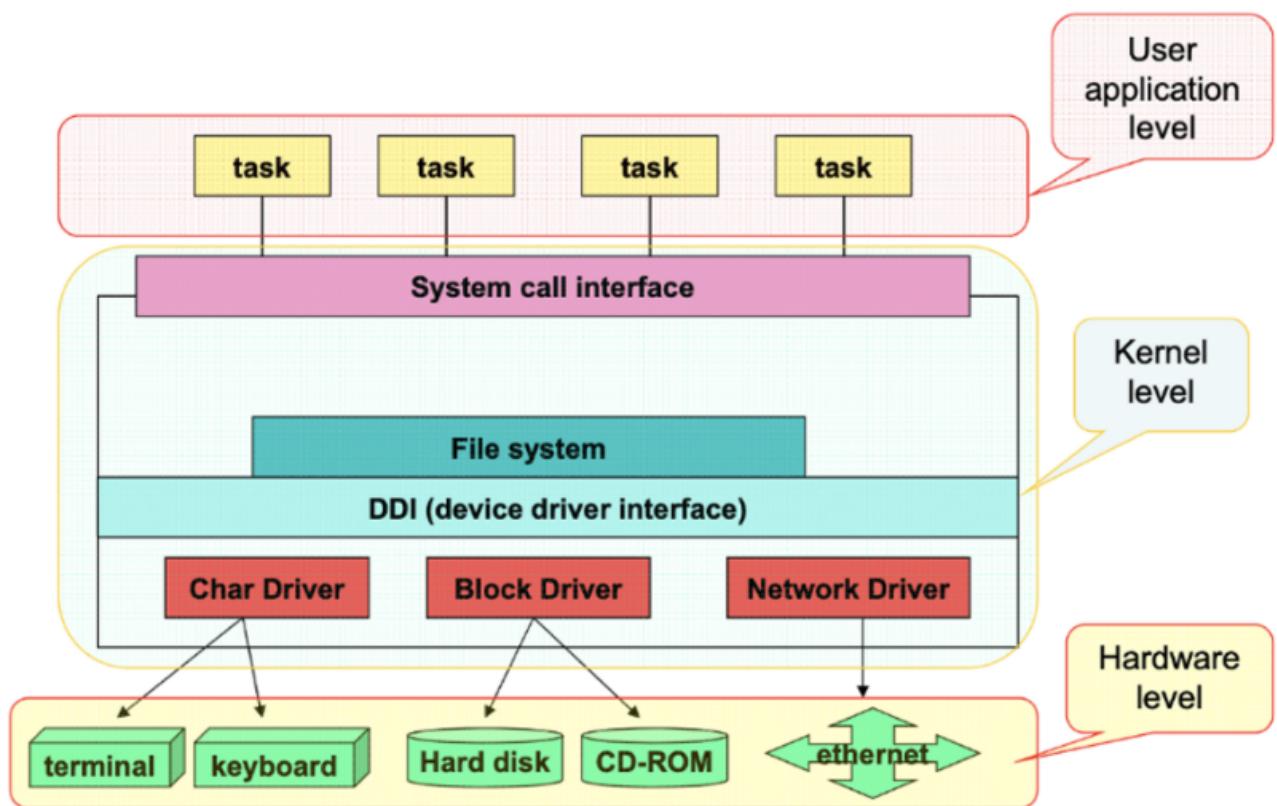
## Navigate space

# Linux kernel

## 1. Linux Device driver

### 1. Types of devices in Linux

Linux kernel diagram including device drivers



### 1.1 Character Devices

- **Description:** Character devices, also known as "char devices," handle data one character at a time.
- **Use Cases:** They are typically used for devices that require sequential access, such as keyboards, mice, serial ports, and more.
- **Access:** Accessed through files in the `/dev` directory. Examples include `/dev/tty` for the terminal, `/dev/null`, and `/dev/random`.

## 1.2 Block Devices

- **Description:** Block devices handle data in blocks, which means they read and write data in fixed-size chunks. This allows for random access to data blocks, enabling users to jump to different locations on the device.
- **Use Cases:** Commonly used for storage devices, such as hard drives, SSDs, and USB flash drives.
- **Access:** Accessed through files in the `/dev` directory. Examples include `/dev/sda` for the first SATA drive, `/dev/nvme0n1` for the first NVMe drive, etc.

## 1.3 Network Devices

- **Description:** Network devices are used to send and receive data packets over a network.
- **Use Cases:** Examples include Ethernet adapters, Wi-Fi cards, and other interfaces that facilitate network communication.
- **Access:** Network interfaces are listed under `/sys/class/net/` or can be seen using the `ip addr` command. They can be interacted with through the utilities like `ifconfig`, `ip`, `netstat`, and others.

## 2. Misc Driver

Misc drivers (miscellaneous drivers) in Linux are a category of device drivers that don't fit well into other standard categories of drivers like network, USB, or block drivers.

They are used for controlling various types of devices that do not belong to a common device class.

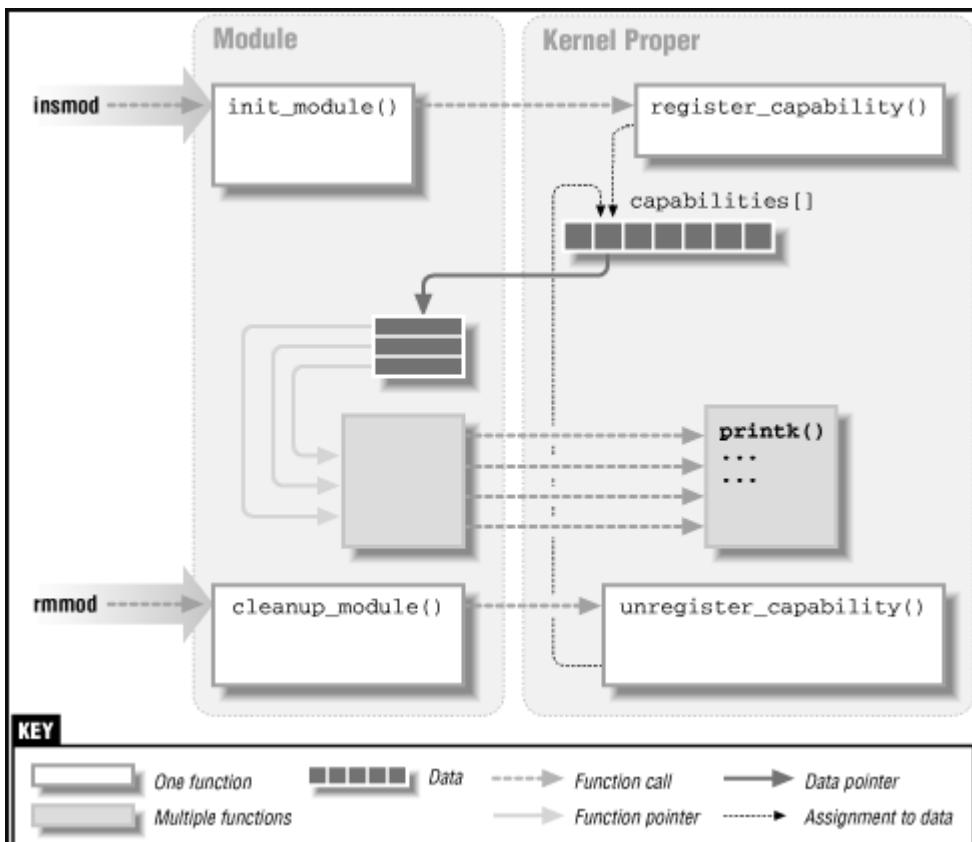
## 3. Module Driver

### 3.1 Explanation

- **Definition:** Module drivers refer to any drivers that are compiled as modules in the Linux kernel. This can include drivers for network interfaces, block devices, USB devices, etc.
- **Characteristics:**
  - **Loadable Kernel Modules (LKMs):** Module drivers can be dynamically loaded into and unloaded from the running kernel, allowing hardware to be added or removed without rebooting the system. (LKM is an option)
  - **Modularity:** This approach supports modularity and extensibility, enabling the kernel to stay lean by loading only the necessary modules for the hardware present.
  - **Maintainability:** Modularization makes it easier to maintain the software. When code is organized into modules, developers can quickly update parts of the application without affecting the entire system.

- **Tools for Management:** Utilities like `modprobe`, `insmod`, and `rmmmod` are used to manage loading and unloading of module drivers.

### 3.2 Diagram showing how LKMs are loaded



## 4. Adding a dummy module driver to the kernel

This section describes how to add a loadable kernel module to the kernel.

### 4.1 Source code

The code in this section registers a device driver controlled by file operations

#### Example code of char driver

##### my\_char.c

```
#include <linux/kernel.h>
#include <linux/init.h>
```

```

#include <linux/module.h>
#include <linux/kdev_t.h>
#include <linux/fs.h>
#include <linux/err.h>
#include <linux/cdev.h>
#include <linux/device.h>
#include <linux/err.h>

dev_t dev = 0;
static struct class *dev_class;
static struct cdev my_cdev;

/*
** Function Prototypes
*/
static int __init my_driver_init(void);
static void __exit my_driver_exit(void);
static int my_open(struct inode *inode, struct file *file);
static int my_release(struct inode *inode, struct file *file);
static ssize_t my_read(struct file *filp, char __user *buf, size_t len, loff_t *off);
static ssize_t my_write(struct file *filp, const char *buf, size_t len, loff_t *off);

static struct file_operations fops =
{
    .owner      = THIS_MODULE,
    .read       = my_read,
    .write      = my_write,
    .open        = my_open,
    .release    = my_release,
};

/*
** This function will be called when we open the Device file
*/
static int my_open(struct inode *inode, struct file *file)
{
    pr_info("YANG Driver Open Function Called...!!!\n");
    return 0;
}

/*
** This function will be called when we close the Device file
*/
static int my_release(struct inode *inode, struct file *file)
{
    pr_info("YANG Driver Release Function Called...!!!\n");
    return 0;
}

/*

```

```

** This function will be called when we read the Device file
*/
static ssize_t my_read(struct file *filp, char __user *buf, size_t len, loff_t *off)
{
    pr_info("YANG Driver Read Function Called....!!!\n");
    return 0;
}

/*
** This function will be called when we write the Device file
*/
static ssize_t my_write(struct file *filp, const char __user *buf, size_t len, loff_t
*off)
{
    pr_info("YANG Driver Write Function Called....!!!\n");
    return len;
}

/*
** Module Init function
*/
static int __init my_driver_init(void)
{
    /*Allocating Major number*/
    if((alloc_chrdev_region(&dev, 0, 1, "my_char_Dev")) <0){
        pr_err("Cannot allocate major number\n");
        return -1;
    }
    pr_info("Major = %d Minor = %d \n",MAJOR(dev), MINOR(dev));

    /*Creating cdev structure*/
    cdev_init(&my_cdev,&fops);

    /*Adding character device to the system*/
    if((cdev_add(&my_cdev,dev,1)) < 0){
        pr_err("Cannot add the device to the system\n");
        goto r_class;
    }

    /*Creating struct class*/
    if(IS_ERR(dev_class = class_create(THIS_MODULE,"my_char_class"))){
        pr_err("Cannot create the struct class\n");
        goto r_class;
    }

    /*Creating device*/
    if(IS_ERR(device_create(dev_class,NULL,dev,NULL,"my_char_device"))){
        pr_err("Cannot create the Device 1\n");
        goto r_device;
    }
    pr_info("YANG Device Driver Insert...Done!!!\n");
    return 0;
}

```

```

r_device:
    class_destroy(dev_class);
r_class:
    unregister_chrdev_region(dev,1);
    return -1;
}

/*
** Module exit function
*/
static void __exit my_driver_exit(void)
{
    device_destroy(dev_class,dev);
    class_destroy(dev_class);
    cdev_del(&my_cdev);
    unregister_chrdev_region(dev, 1);
    pr_info("Device Driver Remove...Done!!!\n");
}

module_init(my_driver_init);
module_exit(my_driver_exit);

MODULE_LICENSE("GPL");
MODULE_AUTHOR("your name");
MODULE_DESCRIPTION("Simple char device driver (File Operations)");
MODULE_VERSION("1.3");

```

### Example code of misc driver

#### my\_misc.c

```

#include <linux/init.h>
#include <linux/module.h>
#include <linux/miscdevice.h>
#include <linux/fs.h>
static int my_misc_device_open(struct inode *inode, struct file *file) {
    pr_info("my_misc_device: open\n");
    return 0;
}

static int my_misc_device_close(struct inode *inodep, struct file *filp) {
    pr_info("my_misc_device: close\n");
    return 0;
}

static const struct file_operations my_misc_fops = {
    .owner = THIS_MODULE,
    .open = my_misc_device_open,
    .release = my_misc_device_close,

```

```

};

static struct miscdevice my_misc_device = {
    .minor = MISC_DYNAMIC_MINOR, // Dynamically allocate a minor number
    .name = "my_misc", // Name of the device file
    .fops = &my_misc_fops, // File operations
};

static int __init my_misc_init(void) {
    int status;

    // Register the misc device
    status = misc_register(&my_misc_device);
    if (status) {
        pr_err("Failed to register misc device\n");
        return status;
    }

    pr_info("my_misc_device registered\n");
    return 0;
}

static void __exit my_misc_exit(void) {
    // Unregister the misc device
    misc_deregister(&my_misc_device);
    pr_info("my_misc_device unregistered\n");
}

module_init(my_misc_init);
module_exit(my_misc_exit);

MODULE_LICENSE("GPL");
MODULE_AUTHOR("Your Name");
MODULE_DESCRIPTION("A simple example of a Linux misc driver");

```

## 4.2 Build the module driver

This section describes how to build a misc driver. Building other drivers follows the same process.

### 4.2.1 Compiling the module along with the kernel

The source(e.g. my\_misc.c) is located in drivers/misc.

#### 4.2.1.1 Optional compile according to Kconfig

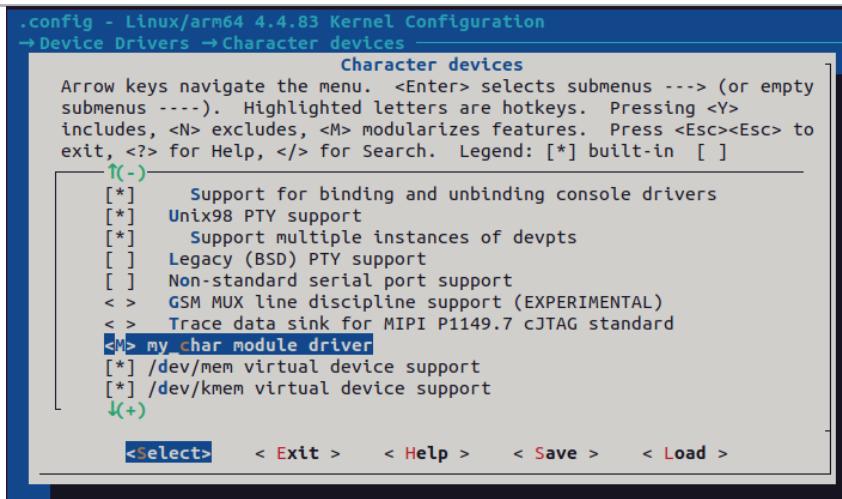
In this case, the module built can be included and excluded optionally via menuconfig

1. Edit `drivers/misc/Kconfig`

```
config MY_MISC
    tristate "my_misc module driver"
    default m
    help "my_misc"
```

## 2. Modify and save menuconfig

```
make ARCH=arm64 menuconfig
```



```
make ARCH=arm64 savedefconfig
cp defconfig ./arch/arm64/configs/s5p6818_bitminer_defconfig
```

## 3. Modify drivers/char/Makefile

```
obj-$(CONFIG_MY_MISC) += my_misc.o
```

### 4.2.1.2 Always compile along with the kernel build

In this case, the module is always built along with the kernel.

Modify `drivers/misc/Makefile`

```
obj-m += my_misc.o
```

### 4.2.2 Manual build

#### 1. Create this initial directory(my\_misc.c)



## 2. Write Makefile

```
obj-m += my_misc.o
KDIR = ~/work/dunfell-bitminer/sources/kernel/kernel-4.4.x
all:
    make -C $(KDIR) M=$(shell pwd) modules ARCH=arm64
clean:
    make -C $(KDIR) M=$(shell pwd) clean
```

## 3. Build

```
sudo make
```

## 4.3 Loading the module driver

Install the built module driver(.ko) to /home/root as described below.

### 4.3.1 If the module was built along with the kernel

1. Find where the module is

```
find / -type f -name "my_misc.ko"
/lib/modules/4.4.83/kernel/drivers/char/my_char.ko
```

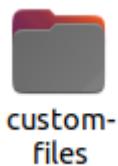
2. Load the module

```
insmod /lib/modules/4.4.83/kernel/drivers/char/my_misc.ko
```

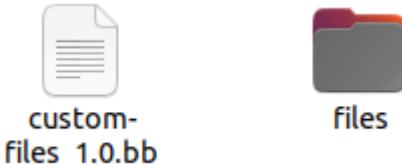
### 4.3.2 If the module was manually built

#### 4.3.2.1 Using the yocto recipe

1. Create a recipe directory in recipes-core(or any recipes- directory)



2. Create a recipe and a `files` directory and add necessary files in the `files` directory



3. Write the recipe

```
DESCRIPTION = "Install custom file to /home/root"
LICENSE = "MIT"

LIC_FILES_CHKSUM = "file://${COREBASE}/meta/
COPYING.MIT;md5=3da9cfbc788c80a0384361b4de20420"

SRC_URI = "file://my_misc.ko"

do_install() {
    install -d ${D}/home/root # Create the directory if it doesn't exist
    install -m 0644 ${WORKDIR}/my_misc.c ${D}/home/root/my_misc.ko
}

# package management system
FILES_${PN} += "/home/root/my_misc.ko"
```

4. Add the recipe to the core recipe

```
IMAGE_INSTALL:append = \
    custom-files \
```

#### 4.3.2.2 Pushing the module to the target board

```
adb push my_misc.ko /home/root
```

```
es/kernel/kernel-4.4.x/drivers/char$ adb push my_char.ko /home/root
my_char.ko: 1 file pushed. 9.9 MB/s (246416 bytes in 0.024s)
```

#### 4.3.2.3 Load the module driver in the target board

```
insmod my_misc.ko
```

```
# insmod my_misc.ko
[ my_misc_device registered
```

`rmmmod my_misc.ko` will remove the inserted module.

### 4.4 Performing operations of the loaded module driver

The registered driver is located in `/dev/<device_name>`. The driver's file operations can be performed using this file.

#### 4.4.1 Source code

This code opens and closes the device driver file `/dev/my_misc` to trigger the file operations defined in **struct file\_operations** within **my\_misc.c**

```
#include <stdio.h>
#include <fcntl.h> // For O_RDWR
#include <unistd.h> // For open(), close()

void main() {
    int fd;
    fd = open("/dev/my_misc", O_RDWR);
    if (fd < 0) {
        perror("Failed to open the device");
        return -1;
    }
    return 0;
}
```

#### 4.4.2 Result

```
my_misc_device: open
my_misc_device: close
```

### Reference

- <https://embedtronix.com/tutorials/linux/device-drivers>

## 2. Device-Tree

### Device tree

#### 1. Understanding the device tree

Refer to the page [ u-boot/1. Device Driver/1 ]



#### 2. How the device tree is loaded by Bootloader

During the boot process, the secondary bootloader loads the DTB into memory. The specific method of loading can vary:

- **Directly from Storage:** The bootloader reads the DTB from its storage location into RAM.
- **Packaged with the Kernel:** In some configurations, the DTB may be appended to the kernel image. The bootloader loads the entire package into memory, and the kernel extracts the DTB.
- **User Selection or Automatic Detection:** In systems with multiple possible hardware configurations, the bootloader may present a selection to the user or automatically detect the hardware configuration to choose the correct DTB.

#### 3. Platform device - Platform driver

- **Direct integration:** Devices managed by platform drivers are tightly integrated with the system hardware.
- **Static Configuration:** Information about the device (like memory addresses, IRQ numbers, etc) is specified within static data like the Device Tree.
- **No Hardware Enumeration:** These devices are typically directly integrated into the motherboard and do not have an enumeration mechanism unlike PCI or USB devices.

### 3.1 How to connect a device to a driver

#### 3.1.1 Add a device(mydevice@1) in the device tree ( s5p6818.dtsi )

```
{  
    model = "nexell soc";  
    compatible = "nexell,s5p6818";  
    #address-cells = <0x1>;  
    #size-cells = <0x1>;  
  
    aliases {  
        serial0 = &serial0;  
        serial1 = &serial1;  
        serial2 = &serial2;  
        serial3 = &serial3;  
        serial4 = &serial4;  
        serial5 = &serial5;  
        i2s0    = &i2s_0;  
        i2s1    = &i2s_1;  
        i2s2    = &i2s_2;  
        spi0    = &spi_0;  
        spi1    = &spi_1;  
        spi2    = &spi_2;  
        i2c0    = &i2c_0;  
        i2c1    = &i2c_1;  
        i2c2    = &i2c_2;  
  
        pinctrl0 = &pinctrl_0;  
    };  
  
    mydevice@1 {  
        compatible = "yang,mydevice";  
        /* Other necessary properties */  
    };
```

### 3.1.2 Source code

The driver code should match the "compatible" (yang,mydevice)

- drivers/platform/my\_platform\_driver.c

```
#include <linux/init.h>
#include <linux/module.h>
#include <linux/platform_device.h>

// Device ID table
static struct of_device_id testdev_of_match[] = {
    { .compatible = "yang,mydevice", },
    {},
};

MODULE_DEVICE_TABLE(of, testdev_of_match);

// Probe function - called when device is detected
static int testdev_probe(struct platform_device *pdev)
{
    printk(KERN_INFO "yang's platform driver probed\n");
    // Device initialization and resource allocation go here
    // For example, mapping device memory:
    // void __iomem *regs = devm_platform_ioremap_resource(pdev, 0);
    // return 0; // Return 0 if device is successfully initialized
}

// Remove function - called when device is removed
static int testdev_remove(struct platform_device *pdev)
{
    printk(KERN_INFO "yang's platform driver removed\n");
    // Cleanup code here, like freeing allocated resources
    return 0;
}

// Platform driver structure
static struct platform_driver testdev_driver = {
    .probe = testdev_probe,
    .remove = testdev_remove,
    .driver = {
        .name = "my_platform_driver",
        .of_match_table = testdev_of_match,
        .owner = THIS_MODULE,
    },
};

// Module initialization
static int __init testdev_init(void)
{
    printk(KERN_INFO "yang's platform driver init\n");
```

```

    return platform_driver_register(&testdev_driver);
}

// Module exit
static void __exit testdev_exit(void)
{
    printk(KERN_INFO "yang's platform driver exit\n");
    platform_driver_unregister(&testdev_driver);
}

module_init(testdev_init);
module_exit(testdev_exit);

MODULE_LICENSE("GPL");
MODULE_AUTHOR("Your Name");
MODULE_DESCRIPTION("Test Platform Driver");

```

- Modify `drivers/platform/Makefile` to add the source to the built list

`obj-y += my_platform_driver.o`

### 3.1.3 Probe

Check the probe log to see if the device has been connected to the driver.

```

dmesg | grep "yang"

# dmesg | grep "yang"
yang's platform driver init
yang's platform driver probed

```

## 4. How to view the device tree in linux

### 4.1 /proc/device-tree/

```

root@s5p6818:~# ls /proc/device-tree/
#address-cells  chosen  gpio_keys  memory  name  nx-v4l2  psci
#size-cells    compatible  i2c@10  model  nexell-ion@0  oscillator  reserved-memory
aliases        cpus      leds     mydevice@1  nx-devfreq  pmu      soc

```

This directory contains directories and files corresponding to nodes and properties defined in the Device Tree

## 4.2 /sys/firmware/devicetree/base/

```
root@s5p6818:~# ls /sys/firmware/devicetree/base/
#address-cells  chosen      gpio_keys      memory      name      nx-v4l2      psci
#size-cells     compatible   i2c@10       model       nexell-ion@0  oscillator  reserved-memory
aliases        cpus        leds          mydevice@1  nx-devfreq pmu           soc
```

This path is like `/proc/device-tree/` but is often preferred because it's more structured and designed to be easier to navigate programmatically.

## 4.3 Reading these directories and files

Reading them allows for obtaining information about the system's hardware configuration, such as peripheral addresses, interrupt numbers, and device parameters.

They can be read as described below.

### 4.3.1 Checking the child nodes of a device

```
ls /sys/firmware/devicetree/base/mydevice@1
```

### 4.3.2 Reading the properties

```
cat /sys/firmware/devicetree/base/mydevice@1/name
cat /sys/firmware/devicetree/base/mydevice@1/compatible
```

## Memory set-up

### 1. Disable the default DRAM setup

First, modify common/fdt\_support.c so that CONFIG\_SYS\_SDRAM\_SIZE in u-boot configs/s5p6818\_bitminer.h does not automatically set up the memory .

`fdt_fixup_memory_banks()` overrides the device tree and sets up the memory.

```
int fdt_fixup_memory_banks(void *blob, u64 start[], u64 size[], int banks)
{
    return 0;
}
```

## 2. Memory size

This section shows how to change the memory size by modifying the device tree.

### 2.1 Before changing the memory size

#### 2.1.1 Device tree(s5p6818-bitminer-common.dtsi)

```
memory {
    device_type = "memory";
    reg = <0x40000000 0x0db00000>;
    /* 0x40000000-0x0db00000 16MB */
}
```

**format:** reg = <start\_address size>

#### 2.1.2 Check the memory size

```
dmesg | grep "memory"
```

```
Memory: 177728K/224256K available (6292K kernel code, 896K rwdta, 3948K rodata, 492K init, 634K bss, 30144K reserved,
```

### 2.2 After changing the memory size

#### 2.2.1 Device tree(s5p6818-bitminer-common.dtsi)

```
memory {
    device_type = "memory";
    reg = <0x40000000 0xab00000>;
    /* 0x40000000-0xab00000 16MB */
}
```

**format:** reg = <start\_address size>

#### 2.2.2 Check the memory size

```
dmesg | grep "memory"
```

```
Memory: 128576K/175104K available (6292K kernel code, 896K rwdta, 3948K rodata, 492K init, 634K bss, 30144K reserved,
```

### 3. Reserved memory

Reserved memory within the Linux kernel refers to portions of memory that are allocated for specific uses or devices and are not available for general-purpose use by the operating system's memory manager.

Below is about how to handle reserved memory using the device tree.

#### 3.1 Before creating reserved memory

##### 3.1.1 Device tree(s5p6818.dtsi)

```
reserved-memory {
    #address-cells = <1>;
    #size-cells = <1>;
    ranges;
};

mydevice@1 {
    compatible = "yang,mydevice";
    /* Other necessary properties */
};
```

##### 3.1.2 Check the memory

```
cat /proc/meminfo
```

```
root@s5p6818:~# cat /proc/meminfo
MemTotal:      194604 kB
```

```
dmesg | grep "memory"
```

```
Memory: 177728K/224256K available (6292K kernel code, 896K rwdta, 3948K rodata, 492K init, 634K bss, 30144K reserved,
```

### 3.2 After creating reserved memory

#### 3.2.1 Device tree(s5p6818.dtsi)

```

reserved-memory {
    #address-cells = <1>;
    #size-cells = <1>;
    ranges;
    my_reserved: buffer@0 {
        reg = <0x40000000 0x2000000>;
    };
};

mydevice@1 {
    compatible = "yang,mydevice";
    #memory-region = <&my_reserved>;
    /* Other necessary properties */
};

```

`memory-region` property can be used to allocate the reserved memory to a device.

#### 3.2.2 Check if the reserved memory was successfully added

```
cat /proc/meminfo
```

```
root@s5p6818:~# cat /proc/meminfo
MemTotal:           174144 kB
```

```
dmesg | grep "memory"
```

```
Memory: 157268K/224256K available (6292K kernel code, 896K rwdata, 3948K rodata, 492K init, 634K bss, 50604K reserved,
```

## 4. CMA(Contiguous Memory Allocator)

CMA allows for the allocation of large contiguous blocks of memory before or after the system has booted. It's particularly useful for devices that need to perform DMA operations requiring contiguous physical memory, such as video frame buffers or hardware that performs direct I/O. CMA helps mitigate the issue of memory fragmentation, making it easier to allocate large contiguous memory regions.

Below is about how to add a CMA memory pool using the device tree.

## 4.1 Device tree(s5p6818.dtsi)

```
reserved-memory {
    #address-cells = <1>;
    #size-cells = <1>;
    ranges;
    my_reserved: buffer@0 {
        #reg = <0x40000000 0x2000000>;
    };
    linux,cma {
        compatible = "shared-dma-pool";
        reusable;
        size = <0x4000000>;
        linux,cma-default;
    };
};
```

## 4.2 Check the added CMA memory

```
dmesg | grep "memory"
[  1.350000] Reserved memory: created CMA memory pool at 0x0000000049800000, size 64 MiB
```

## Reference

- Documentation/devicetree/bindings/reserved-memory/reserved-memory.txt

## 3. Driver sequence

### 1. Kernel initialization

The Linux kernel initialization is a highly structured process that ensures system components and drivers are started in a specific order, allowing dependencies to be resolved correctly.

The initialization sequence consists of several levels.

#### 1.1 Kernel initialization level

Each phase is called by **initcall()**

1. **pure**: Architecture-specific, dealing with setting up essential hardware configurations.

2. **core**: Involves setting up essential kernel services and infrastructure, scheduling, interrupt handling, and basic memory management.
3. **postcore**: Initializes additional core subsystems that depend on the very basic services set up during the core phase.
4. **arch**: Since the Linux kernel supports multiple hardware architectures (such as x86, ARM, MIPS, etc.), this phase customizes the initialization process to the specific requirements of the current architecture.
5. **subsys**: The subsystems initializes various kernel subsystems that are not directly tied to the core kernel functionality. This can include driver frameworks, networking, filesystem support, and other subsystems that provide higher-level services.
6. **fs**: The filesystem initialization phase sets up the kernel's filesystem infrastructure, allowing the kernel to access file systems on disk, which is critical for the rest of the system's operation.
7. **rootfs**: The root filesystem is mounted during this phase.
8. **device**: This stage involves the initialization of device drivers and the device model, which allows the kernel to interact with the hardware components of the system.
9. **late**: The late initialization phase is the final step, handling any remaining initialization. This might include starting up user-space applications and services that are essential for the system's operation.

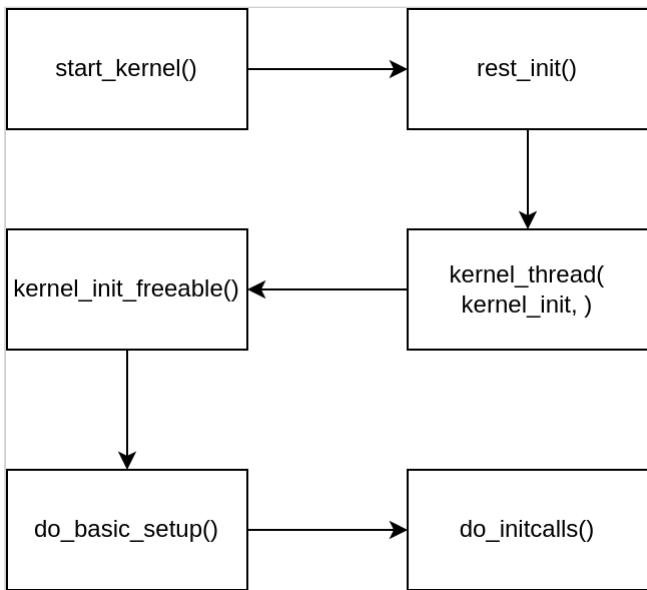
## 1.2 Kernel initialization process

### 1.2.1 Flow of function calls

head.S runs init/main.c :**start\_kernel()**

```
kernel-4.4.x > arch > arm64 > kernel > asm head.S
#ifndef CONFIG_RANDOMIZE_BASE
    bl kasan_early_init
#endif
    #ifdef CONFIG_RANDOMIZE_BASE
        tst x23, ~(MIN_KIMG_ALIGN - 1) // already running randomized?
        bne    of
        mov x0, x21                // pass FDT address in x0
        mov x1, x23                // pass modulo offset in x1
        bl kaslr_early_init        // parse FDT for KASLR options
        cbz x0, of                // KASLR disabled? just proceed
        orr x23, x23, x0          // record KASLR offset
        ret x28                  // we must enable KASLR, return
        | | | | | | | |           // to __enable_mm()
    o:
#endif
    #endif
    *+ b start_kernel
ENDPROC(__mmap_switched)
```

**start\_kernel()** initializes various components and leads to **do\_initcalls()**, which runs each init stage.



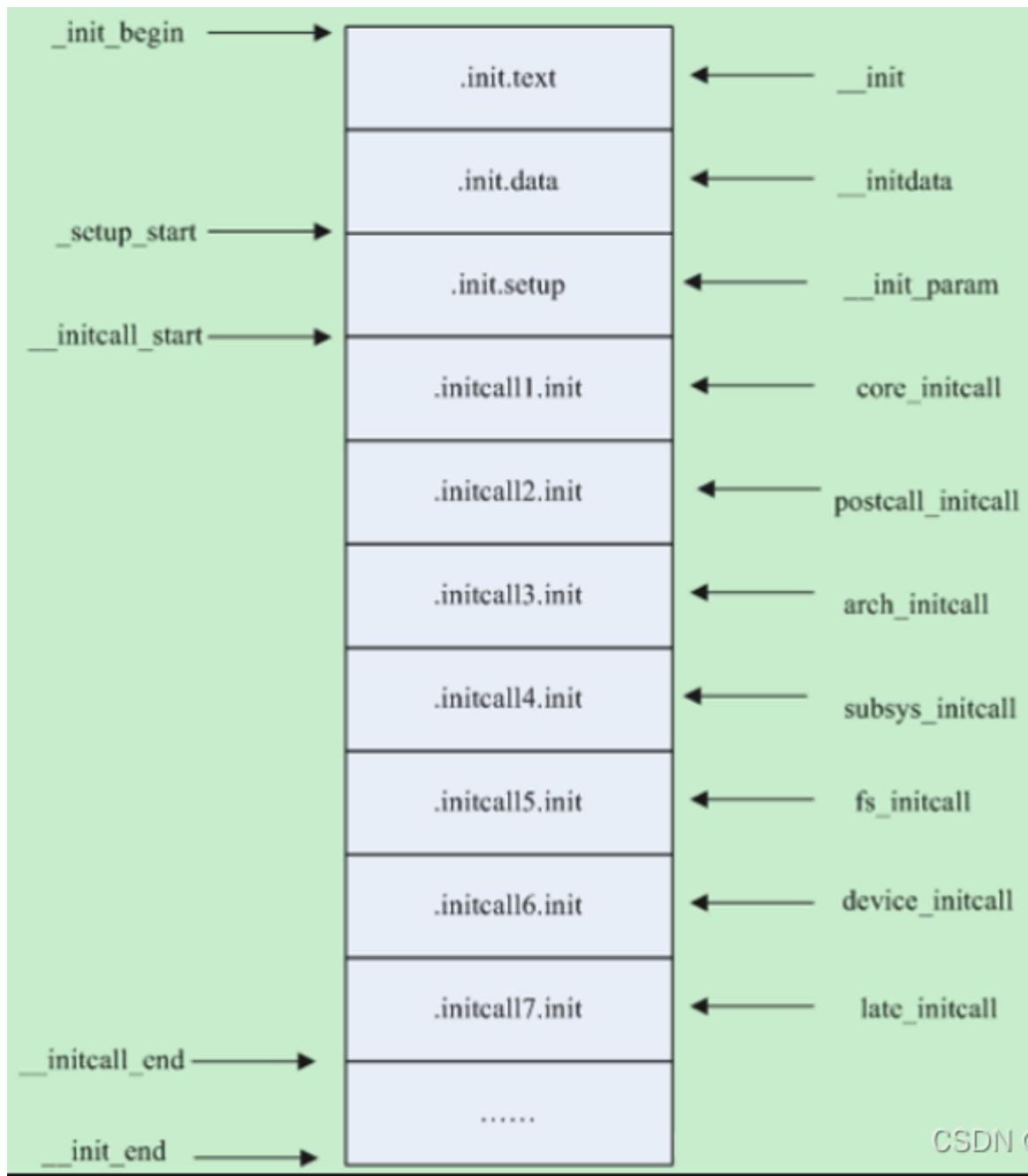
## 1.2.2 Registration of initcalls

Macros defined in `<include/linux/init.h>` register function pointers in specific sections of the kernel binary so that they can be run in specific init stages.

```

#define pure_initcall(fn)          __define_initcall(fn, 0)
#define core_initcall(fn)          __define_initcall(fn, 1)
#define core_initcall_sync(fn)     __define_initcall(fn, 1s)
#define postcore_initcall(fn)      __define_initcall(fn, 2)
#define postcore_initcall_sync(fn) __define_initcall(fn, 2s)
#define arch_initcall(fn)          __define_initcall(fn, 3)
#define arch_initcall_sync(fn)     __define_initcall(fn, 3s)
#define subsys_initcall(fn)        __define_initcall(fn, 4)
#define subsys_initcall_sync(fn)   __define_initcall(fn, 4s)
#define fs_initcall(fn)            __define_initcall(fn, 5)
#define fs_initcall_sync(fn)       __define_initcall(fn, 5s)
#define rootfs_initcall(fn)        __define_initcall(fn, rootfs)
#define device_initcall(fn)        __define_initcall(fn, 6)
#define device_initcall_sync(fn)   __define_initcall(fn, 6s)
#define late_initcall(fn)          __define_initcall(fn, 7)
#define late_initcall_sync(fn)     __define_initcall(fn, 7s)
  
```

The linker script(``vmlinux.lds``) organizes these special sections so that the function pointers are registered well in their correct section.



Registered functions can be seen using '`nm vmlinux | grep`'

```
es/kernel/kernel-4.4.x$ nm vmlinux | grep mydevice_debug_init
fffff8008add530 d __initcall_mydevice_debug_init2
fffff8008aad9f4 t mydevice_debug_init
```

During the kernel's boot process, `do_initcalls()` iterates over these sections and executes the function pointers contained within them.

```

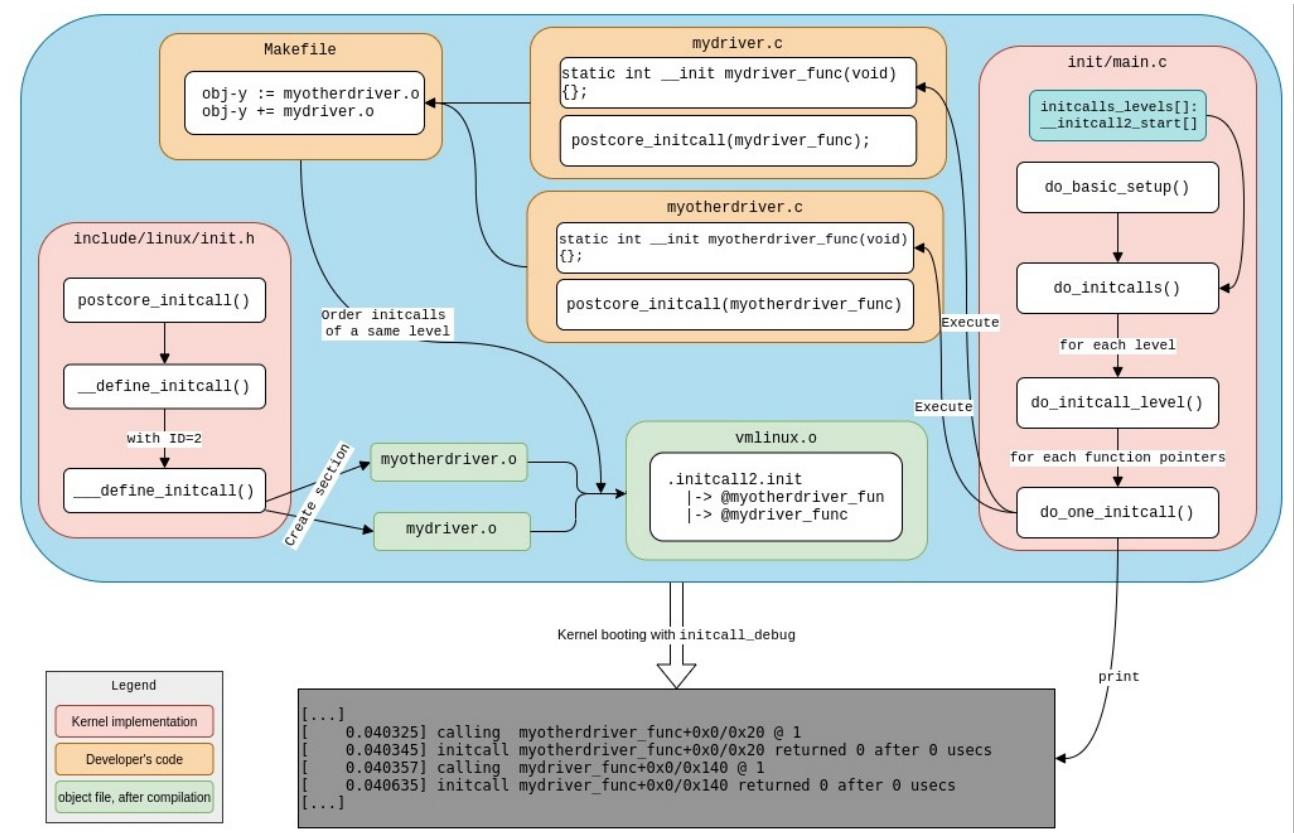
static void __init do_initcalls(void)
{
    int level;

#ifdef CONFIG_INITCALLS_THREAD
    for (level = 0; level < CONFIG_INITCALLS_THREAD_LEVEL; level++)
        do_initcall_level(level);

    init_thread_level = level;
    kthread_run(do_initcalls_kth,
                (void *)&init_thread_level, "initcalls:thread");
#else
    for (level = 0; level < ARRAY_SIZE(initcall_levels) - 1; level++)
        do_initcall_level(level);
#endif
}

```

### 1.3 Diagram showing how modules are loaded in the kernel initialization



## 2. Module driver

### 2.1 Registration and loading of a module driver

- `module_init()` registers a module driver

```
static int __init misc_init(void)
{
    int error;

    error = misc_register(&etx_misc_device);
    if (error) {
        pr_err("misc_register failed!!!\n");
        return error;
    }

    pr_info("misc_register init done!!!\n");
    return 0;
}
module_init(misc_init)
```

- `module_init()` is defined in `include/linux/module.h`

```
#define module_init(x) __initcall(x);
```

- `__initcall()` is defined in `include/linux/init.h`

```
#define core_initcall(fn) __define_initcall(fn, 1)
#define core_initcall_sync(fn) __define_initcall(fn, 1s)
#define postcore_initcall(fn) __define_initcall(fn, 2)
#define postcore_initcall_sync(fn) __define_initcall(fn, 2s)
#define arch_initcall(fn) __define_initcall(fn, 3)
#define arch_initcall_sync(fn) __define_initcall(fn, 3s)
#define subsys_initcall(fn) __define_initcall(fn, 4)
#define subsys_initcall_sync(fn) __define_initcall(fn, 4s)
#define fs_initcall(fn) __define_initcall(fn, 5)
#define fs_initcall_sync(fn) __define_initcall(fn, 5s)
#define rootfs_initcall(fn) __define_initcall(fn, rootfs)
#define device_initcall(fn) __define_initcall(fn, 6)
#define device_initcall_sync(fn) __define_initcall(fn, 6s)
#define late_initcall(fn) __define_initcall(fn, 7)
#define late_initcall_sync(fn) __define_initcall(fn, 7s)

#define __initcall(fn) device_initcall(fn)
```

This shows `module_init()` registers a module driver in the **device** phase.

If it is a built-in module, it is automatically loaded in the **device phase at boot time**.

## 2.2 Registering a module driver in 'postcore'

- Registering the driver in the **late** phase

```
late_initcall(mydevice_debug_init);
```

- Registering the driver in the **postcore** phase

```
postcore_initcall(mydevice_debug_init);
```

### 2.2.1 Check by time

The pictures below show how device driver messages are recorded at different times on different kernel initialization levels.

- Device driver registered in the '**late**' phase using '**late\_initcall()**'.

```
root@s5p6818:~# dmesg | grep yang
[    2.488000] ===yang's misc device[mydevice_debug_init][L:34]===
[    2.492000] ===yang's misc device[mydevice_probe][L:19]
```

- Device driver registered in the '**postcore**' phase using '**postcore\_initcall()**'.

```
root@s5p6818:~# dmesg | grep yang
[    0.184000] ===yang's misc device[mydevice_debug_init][L:34]===
[    0.208000] ===yang's misc device[mydevice_probe][L:19]
```

The driver init is recorded at an earlier time on **postcore** as shown above.

### 2.2.2 Check by modifying do\_initcalls()

Add a print function to display the current initcall level

```

static void __init do_initcalls(void)
{
    int level;

#ifdef CONFIG_INITCALLS_THREAD
    for (level = 0; level < CONFIG_INITCALLS_THREAD_LEVEL; level++)
        do_initcall_level(level);

    init_thread_level = level;
    kthread_run(do_initcalls_kth,
                (void *)&init_thread_level, "initcalls:thread");
#else
    for (level = 0; level < ARRAY_SIZE(initcall_levels) - 1; level++) {
        pr_info("# Initcall level: %s-----\n", initcall_level_names[level]);
        do_initcall_level(level);
    }
#endif
}

```

You can see the driver is loaded in the **postcore** phase

```

[ 0.184000] # Initcall level: postcore-----
[ 0.184000] ===yang's misc device[mydevice_debug_init][L:34]===

```

## Reference

- <https://kkslinuxinfo.wordpress.com/2015/12/11/kernel-initialization/>
- [https://blog.csdn.net/weixin\\_47491758/article/details/131157608](https://blog.csdn.net/weixin_47491758/article/details/131157608)
- <https://www.collabora.com/news-and-blog/blog/2020/09/25/initcalls-part-2-digging-into-implementation/>

## 4. Interrupt

### 1. Polling

Each case is checked by one by one in an infinite loop. This busy-waiting causes overheads. This is why the interrupt is used instead.

#### Polling

```

while True:

```

```

if request==0:
    response0()
elif request==1:
    response1()
else:
    response2()

```

## 2. About interrupt

An **interrupt** is a request for the processor to temporarily halt the current task and switch to another task with a higher priority.

### 2.1 Types of the interrupt

- **external interrupt(hardware interrupt)** happens as a result of outside interference such as from the user or from the peripherals. It functions as a notifier. (Considered a type of exception)
- **internal interrupt**, also called Trap, happens when wrong instructions or data are used. (exceptions like divided by zero, overflow)
- **software interrupt(system call)**: is a type of interrupt that is triggered by a specific instruction in a program, rather than by an external event or hardware malfunction. It's a mechanism for a program to interrupt the current process flow and request a service from the operating system.  
A program running in user mode needs to send a system call to the operating system to access system resources and perform tasks in kernel mode.

### 2.2 Steps of an interrupt

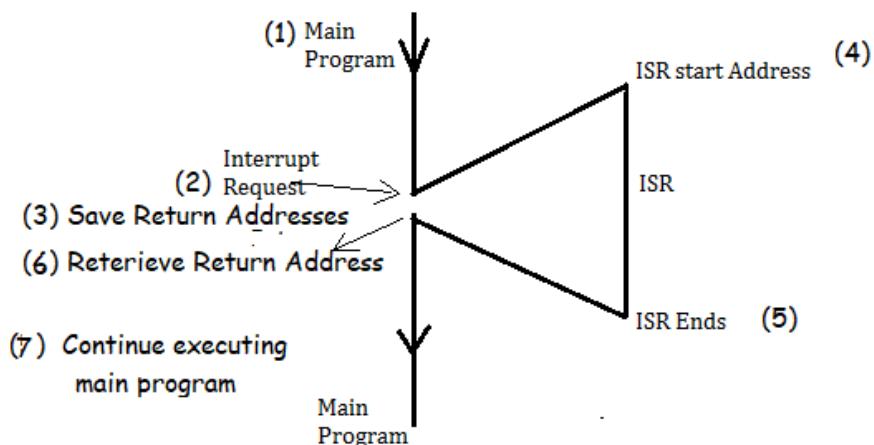


Figure 3.2 Interrupt Cycle

1. An interrupt occurs
2. Current program status is saved onto a stack
3. Jump to the interrupt vector

4. The ISR is executed
5. Jump to the previous program

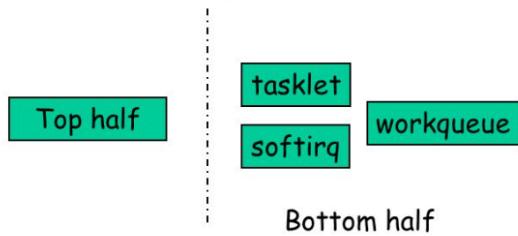
### 2.3. ISR(Interrupt Service Routine)

ISR refers to a function that executes in response to an interrupt signal.

ISR handling in Linux is divided into top-half and bottom-half to balance the need for immediate response to interrupts with the need to perform more complex processing without compromising system responsiveness.

## Linux Interrupt Handler Structure

- Top half (th) and bottom half (bh)
  - Top-half: do minimum work and return (ISR)
  - Bottom-half: deferred processing (softirqs, tasklets, workqueues)



#### 2.3.1 Top-half

- The top half refers to the initial response(interrupt handler) to an interrupt.
- It is supposed to perform minimal work such as acknowledging the interrupt, reading or writing the minimal necessary data to ensure the system can resume its operation as quickly as possible.
- It defers the rest of the time-consuming work to the bottom half.
- Executed in the interrupt context, thus cannot sleep.

#### 2.3.2 Bottom-half

This is where the deferred work from the interrupt is handled and can be done in various ways as listed below:

Feature	SoftIRQ	Tasklet	Workqueue	Threaded IRQ
---------	---------	---------	-----------	--------------

<b>Deferred work</b>	Deferred work is executed in a SoftIRQ statically defined at compile time	Deferred work is put in a queue	Deferred work is put in a queue	Deferred work is executed in the context of a kernel thread where the kernel synchronization mechanisms are available
<b>Context</b>	Interrupt(atomic) context; cannot sleep	Interrupt(atomic) context; cannot sleep	Process context; can sleep	Process context; can sleep
<b>Use Case</b>	Suitable for high-speed and low-latency tasks e.g. Network packet processing, timer updates	Non-blocking operations without the need for waiting e.g. Device driver deferred processing	Long-running jobs that might need to wait e.g. Filesystem operations, scheduled maintenance tasks	Tasks that need the standard synchronization APIs of the kernel like the mutex
<b>Note</b>	Rarely used since there is already the Tasklet			Simple code, easy to use

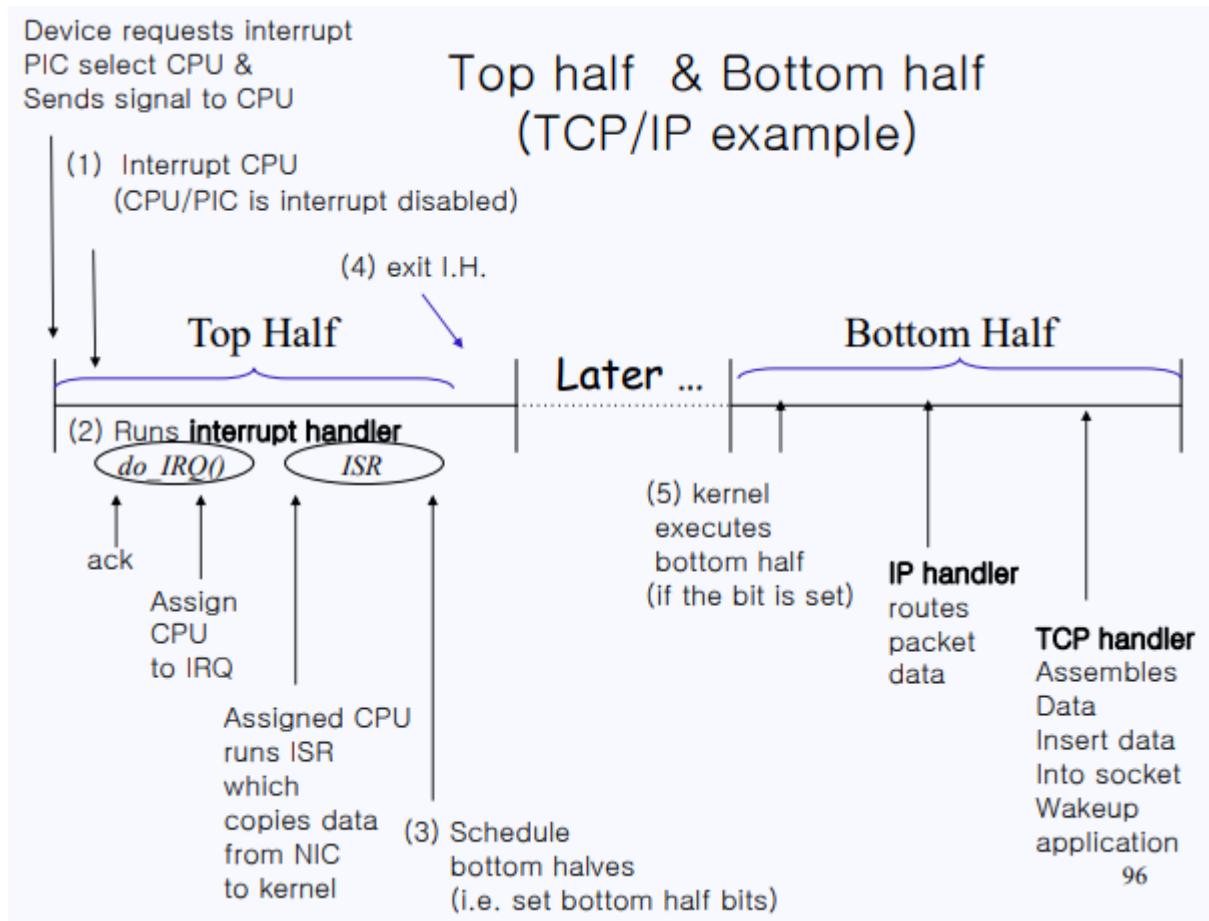
### 2.3.2.1 Difference between delay() and sleep()

<linux/delay.h> has delay() and sleep() that allow pausing

- **delay()**: busy-waiting(spin-lock) where the processor remains active but does nothing productive while waiting for the specified duration to elapse.
  - Bottom-half mechanisms that cannot sleep can be paused by **delay()**
- **sleep()**: causes the thread to yield the CPU and enter a non-running state for the specified duration. The scheduler will not allocate CPU to the sleeping process, allowing other processes to run.

### 2.3.3 Example

Example of the top-half & bottom-half in TCP/IPS



### 3. Interrupt practice

This section describes several ways to register an interrupt that prints a log and toggles LEDs on button press.

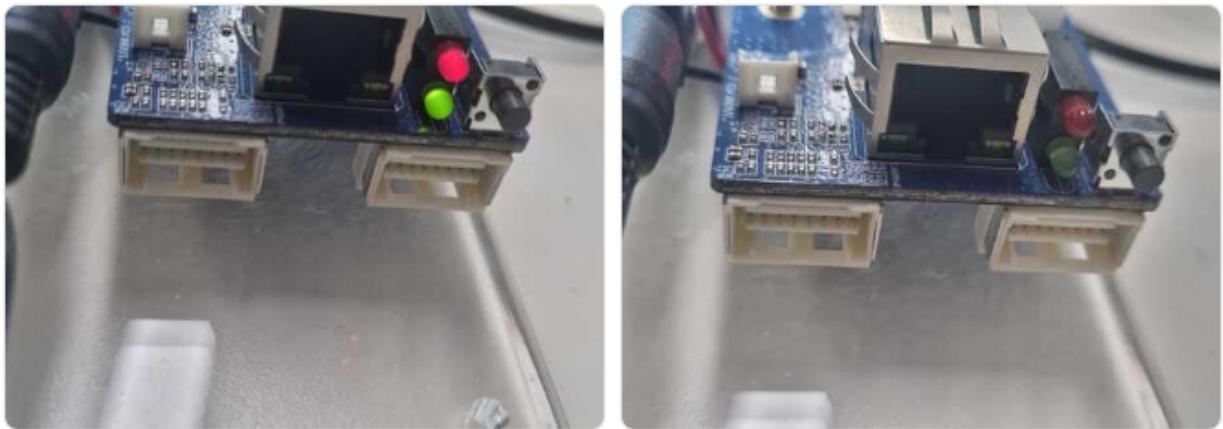
In all the examples below, on button presses:

- "Button pressed!" is printed

```

root@s5p6818:~# [ 28.368000] Button pressed!
[ 29.132000] Button pressed!
[ 29.772000] Button pressed!
[ 30.116000] Button pressed!
[ 31.948000] Button pressed!
    
```

- And LEDs are toggled



### 3.1 Directly in the top-half

In this practice, an interrupt is registered that processes its work in the top-half without passing it to the bottom-half.

In **btn\_irq\_init()**

1. GPIOs are initialized
2. **gpio\_to\_irq()** gets the IRQ number that corresponds to the button GPIO.
3. **request\_irq()** registers an interrupt that executes **btn\_irq\_handler()** in reaction to button press.

#### top\_half.c

```
#include <linux/module.h>
#include <linux/gpio.h>
#include <linux/interrupt.h>
#include <linux/kernel.h>

#define button 3 // Example GPIO number
#define LED1 4
#define LED2 5
#define ON 0
#define OFF 1

static unsigned int irq_number;
unsigned int gpio_value = OFF;

// Interrupt handler function(top-half)
static irqreturn_t btn_irq_handler(int irq, void *dev_id)
{
    gpio_value = gpio_get_value(LED1);
    gpio_set_value(LED1, !gpio_value);
    gpio_set_value(LED2, !gpio_value);
    // gpio_value != gpio_value;
```

```

    pr_info("Button pressed!\n");
    return IRQ_HANDLED;
}

static int __init btn_irq_init(void)
{
    // Request GPIOs
    int result_btn, result1, result2, result_irq;
    result_btn = gpio_request(button, "sysfs");
    result1 = gpio_request(LED1, "sysfs");
    result2 = gpio_request(LED2, "sysfs");
    if (result_btn || result1 || result2) {
        pr_info("Cannot request the GPIO\n");
        return 0;
    }
    // Set the direction of button GPIO
    gpio_direction_input(button);
    // Get the IRQ number for our GPIO
    irq_number = gpio_to_irq(button);
    if (irq_number < 0) {
        printk(KERN_INFO "GPIO to IRQ mapping failed\n");
        return irq_number;
    }
    // Request the IRQ line
    result_irq = request_irq(irq_number, btn_irq_handler, IRQF_TRIGGER_FALLING,
"btn_irq", NULL);
    if (result_irq) {
        printk(KERN_INFO "IRQ request failed\n");
        return result_irq;
    }
    return 0;
}

static void __exit btn_irq_exit(void) {
    free_irq(irq_number, NULL);
    gpio_free(button);
    gpio_free(LED1);
    gpio_free(LED2);
}

module_init(btn_irq_init);
module_exit(btn_irq_exit);

MODULE_LICENSE("GPL");
MODULE_AUTHOR("Your Name");
MODULE_DESCRIPTION("A simple Linux driver for a GPIO interrupt");
MODULE_VERSION("0.1");

```

## 3.2 Using softIRQ

This method registers an interrupt that passes its work to the bottom-half using softIRQ.

1. Add a softIRQ(**YANG\_SOFTIRQ**) in interrupt.h

```
enum
{
    HI_SOFTIRQ=0,
    TIMER_SOFTIRQ,
    NET_TX_SOFTIRQ,
    NET_RX_SOFTIRQ,
    BLOCK_SOFTIRQ,
    BLOCK_IOPOLL_SOFTIRQ,
    TASKLET_SOFTIRQ,
    SCHED_SOFTIRQ,
    HRTIMER_SOFTIRQ, /* Used for numbering
    RCU_SOFTIRQ,      /* Preemptible RCU
    NR_SOFTIRQS,
    YANG_SOFTIRQ
};
```

1. **open\_softirq()** in **btn\_irq\_init()** assigns **bottom\_half()** to the softIRQ "YANG\_SOFTIRQ".
2. **raise\_softirq()** in **btn\_irq\_handler()** triggers the target softIRQ.

### softIRQ.c

```
#include <linux/module.h>
#include <linux/gpio.h>
#include <linux/interrupt.h>
#include <linux/kernel.h>
#include <linux/slab.h> // kmalloc()

#define button 3 // Example GPIO number
#define LED1 4
#define LED2 5
#define ON 0
#define OFF 1

static unsigned int irq_number;
unsigned int gpio_value = OFF;

static void bottom_half(struct softirq_action *action){
    gpio_value = gpio_get_value(LED1);
    gpio_set_value(LED1, !gpio_value);
    gpio_set_value(LED2, !gpio_value);
    // gpio_value != gpio_value;
    pr_info("Button pressed!\n");
}

// Interrupt handler function(top-half)
static irqreturn_t btn_irq_handler(int irq, void *dev_id)
```

```

{
    raise_softirq(YANG_SOFTIRQ);
    return IRQ_HANDLED;
}

static int __init btn_irq_init(void)
{
    // Request GPIOs
    int result_btn, result1, result2, result_irq;
    result_btn = gpio_request(button, "sysfs");
    result1 = gpio_request(LED1, "sysfs");
    result2 = gpio_request(LED2, "sysfs");
    if (result_btn || result1 || result2) {
        pr_info("Cannot request the GPIO\n");
        return 0;
    }
    // Set the direction of button GPIO
    gpio_direction_input(button);
    // Get the IRQ number for our GPIO
    irq_number = gpio_to_irq(button);
    if (irq_number < 0) {
        printk(KERN_INFO "GPIO to IRQ mapping failed\n");
        return irq_number;
    }
    // Assign bottom half() to the softIRQ "YANG_SOFTIRQ"
    open_softirq(YANG_SOFTIRQ, bottom_half);
    // Request the IRQ line
    result_irq = request_irq(irq_number, btn_irq_handler, IRQF_TRIGGER_FALLING,
    "btn_irq", NULL);
    if (result_irq) {
        printk(KERN_INFO "IRQ request failed\n");
        return result_irq;
    }
    return 0;
}

static void __exit btn_irq_exit(void) {
    free_irq(irq_number, NULL);
    gpio_free(button);
    gpio_free(LED1);
    gpio_free(LED2);
}

module_init(btn_irq_init);
module_exit(btn_irq_exit);

MODULE_LICENSE("GPL");
MODULE_AUTHOR("Your Name");
MODULE_DESCRIPTION("A simple Linux driver for a GPIO interrupt");
MODULE_VERSION("0.1");

```

### 3.3 Using tasklet

This method registers an interrupt that passes its work to the bottom-half using tasklet.

1. **DECLARE\_TASKLET()** initializes a **tasklet\_struct** named tasklet using **bottom\_half()** statically at compile time.

- `interrupt.h`

```
#define DECLARE_TASKLET(name, func, data) \  
struct tasklet_struct name = { NULL, 0, ATOMIC_INIT(0), func, data }
```

1. A tasklet is placed into one queue out of two, depending on the priority (1. high priority 2. normal priority).
  - **tasklet\_schedule()** puts the **tasklet\_struct** in the normal priority queue.
  - **tasklet\_hi\_schedule()** will put the **tasklet\_struct** in the high priority queue.
2. **tasklet\_kill()** in **btn\_irq\_exit()** kills the tasklet.

The code below demonstrates the static allocation of a tasklet. Dynamic allocation is also possible following the same method described in section **3.4.2**.

#### tasklet.c

```
#include <linux/module.h>
#include <linux/gpio.h>
#include <linux/interrupt.h>
#include <linux/kernel.h>
#include <linux/workqueue.h>
#include <linux/slab.h> // kmalloc()

#define button 3 // Example GPIO number
#define LED1 4
#define LED2 5
#define ON 0
#define OFF 1

static unsigned int irq_number;
unsigned int gpio_value = OFF;

static void bottom_half(const char *data){
    gpio_value = gpio_get_value(LED1);
    gpio_set_value(LED1, !gpio_value);
    gpio_set_value(LED2, !gpio_value);
    // gpio_value != gpio_value;
    pr_info("Button pressed! %s\n", data);
}
```

```

DECLARE_TASKLET(tasklet, bottom_half, "hello");

// Interrupt handler function(top-half)
static irqreturn_t btn_irq_handler(int irq, void *dev_id)
{
    tasklet_schedule(&tasklet);
    return IRQ_HANDLED;
}

static int __init btn_irq_init(void)
{
    // Request GPIOs
    int result_btn, result1, result2, result_irq;
    result_btn = gpio_request(button, "sysfs");
    result1 = gpio_request(LED1, "sysfs");
    result2 = gpio_request(LED2, "sysfs");
    if (result_btn || result1 || result2) {
        pr_info("Cannot request the GPIO\n");
        return 0;
    }
    // Set the direction of button GPIO
    gpio_direction_input(button);
    // Get the IRQ number for our GPIO
    irq_number = gpio_to_irq(button);
    if (irq_number < 0) {
        printk(KERN_INFO "GPIO to IRQ mapping failed\n");
        return irq_number;
    }
    // Request the IRQ line
    result_irq = request_irq(irq_number, btn_irq_handler, IRQF_TRIGGER_FALLING,
                           "btn_irq", NULL);
    if (result_irq) {
        printk(KERN_INFO "IRQ request failed\n");
        return result_irq;
    }
    return 0;
}

static void __exit btn_irq_exit(void) {
    free_irq(irq_number, NULL);
    gpio_free(button);
    gpio_free(LED1);
    gpio_free(LED2);
    tasklet_kill(&tasklet);
}

module_init(btn_irq_init);
module_exit(btn_irq_exit);

MODULE_LICENSE("GPL");
MODULE_AUTHOR("Your Name");
MODULE_DESCRIPTION("A simple Linux driver for a GPIO interrupt");

```

```
MODULE_VERSION("0.1");
```

## 3.4 Using workqueue

This method registers an interrupt that passes its work to the bottom-half using workqueue.

### 3.4.1 Static allocation of work\_struct and using the kernel-global workqueue

1. **DECLARE\_WORK()** initializes a **work\_struct** named work using **bottom\_half()** statically at compile time.

- `workqueue.h`

```
#define DECLARE_WORK(n, f) \
    struct work_struct n = __WORK_INITIALIZER(n, f)
```

1. **schedule\_work()** puts the **work\_struct** in the kernel-global workqueue.

#### **workqueue.c**

```
#include <linux/module.h>
#include <linux/gpio.h>
#include <linux/interrupt.h>
#include <linux/kernel.h>
#include <linux/workqueue.h>
#include <linux/slab.h> // kmalloc()

#define button 3 // Example GPIO number
#define LED1 4
#define LED2 5
#define ON 0
#define OFF 1

static unsigned int irq_number;
unsigned int gpio_value = OFF;

static void bottom_half(struct work_struct *work) {
    gpio_value = gpio_get_value(LED1);
    gpio_set_value(LED1, !gpio_value);
    gpio_set_value(LED2, !gpio_value);
    // gpio_value != gpio_value;
    pr_info("Button pressed!\n");
}

// Initialize a work_struct named work using bottom_half() statically at compile
time.
```

```

DECLARE_WORK(work, bottom_half);

// Interrupt handler function(top-half)
static irqreturn_t btn_irq_handler(int irq, void *dev_id)
{
    // Put the work in the kernel-global workqueue
    schedule_work(&work);
    return IRQ_HANDLED;
}

static int __init btn_irq_init(void)
{
    // Request GPIOs
    int result_btn, result1, result2, result_irq;
    result_btn = gpio_request(button, "sysfs");
    result1 = gpio_request(LED1, "sysfs");
    result2 = gpio_request(LED2, "sysfs");
    if (result_btn || result1 || result2) {
        pr_info("Cannot request the GPIO\n");
        return 0;
    }
    // Set the direction of button GPIO
    gpio_direction_input(button);
    // Get the IRQ number for our GPIO
    irq_number = gpio_to_irq(button);
    if (irq_number < 0) {
        printk(KERN_INFO "GPIO to IRQ mapping failed\n");
        return irq_number;
    }
    // Request the IRQ line
    result_irq = request_irq(irq_number, btn_irq_handler, IRQF_TRIGGER_FALLING,
                           "btn_irq", NULL);
    if (result_irq) {
        printk(KERN_INFO "IRQ request failed\n");
        return result_irq;
    }
    return 0;
}

static void __exit btn_irq_exit(void) {
    free_irq(irq_number, NULL);
    gpio_free(button);
    gpio_free(LED1);
    gpio_free(LED2);
}

module_init(btn_irq_init);
module_exit(btn_irq_exit);

MODULE_LICENSE("GPL");
MODULE_AUTHOR("Your Name");
MODULE_DESCRIPTION("A simple Linux driver for a GPIO interrupt");

```

```
MODULE_VERSION("0.1");
```

### 3.4.2 Dynamic allocation of work\_struct at runtime

1. In **btn\_irq\_init()**
  - a. **create\_workqueue()** creates a workqueue
  - b. **kmalloc()** allocates **work\_struct**, **INIT\_WORK()** initializes it. (The use of **GFP\_ATOMIC** in **kmalloc()** means that sleep is not allowed. **GFP\_KERNEL** means sleep is allowed.)
2. **queue\_work()** puts the **work\_struct** in the custom workqueue.
3. **work\_struct** is freed using **kfree()** in **btn\_irq\_exit()**.

#### workqueue.c

```
#include <linux/module.h>
#include <linux/gpio.h>
#include <linux/interrupt.h>
#include <linux/kernel.h>
#include <linux/workqueue.h>
#include <linux/slab.h> // kmalloc()

#define button 3 // Example GPIO number
#define LED1 4
#define LED2 5
#define ON 0
#define OFF 1

static unsigned int irq_number;
unsigned int gpio_value = OFF;
static struct work_struct *work;
static struct workqueue_struct *workqueue;

static void bottom_half(struct work_struct *work) {
    gpio_value = gpio_get_value(LED1);
    gpio_set_value(LED1, !gpio_value);
    gpio_set_value(LED2, !gpio_value);
    // gpio_value != gpio_value;
    pr_info("Button pressed!\n");
}

// Interrupt handler function(top-half)
static irqreturn_t btn_irq_handler(int irq, void *dev_id)
{
    // Put the work in the custom workqueue
    queue_work(workqueue, work);
    return IRQ_HANDLED;
}
```

```

static int __init btn_irq_init(void)
{
    // Request GPIOs
    int result_btn, result1, result2, result_irq;
    result_btn = gpio_request(button, "sysfs");
    result1 = gpio_request(LED1, "sysfs");
    result2 = gpio_request(LED2, "sysfs");
    if (result_btn || result1 || result2) {
        pr_info("Cannot request the GPIO\n");
        return 0;
    }
    // Set the direction of button GPIO
    gpio_direction_input(button);
    // Create a workqueue
    workqueue = create_workqueue("own_wq");
    // Allocate work_struct
    work = (struct work_struct *)kmalloc(sizeof(struct work_struct), GFP_ATOMIC);
    // Initialize the work_struct using bottom_half()
    INIT_WORK(work, bottom_half);
    // Get the IRQ number for our GPIO
    irq_number = gpio_to_irq(button);
    if (irq_number < 0) {
        printk(KERN_INFO "GPIO to IRQ mapping failed\n");
        return irq_number;
    }
    // Request the IRQ line
    result_irq = request_irq(irq_number, btn_irq_handler, IRQF_TRIGGER_FALLING,
                           "btn_irq", NULL);
    if (result_irq) {
        printk(KERN_INFO "IRQ request failed\n");
        return result_irq;
    }
    return 0;
}

static void __exit btn_irq_exit(void) {
    free_irq(irq_number, NULL);
    gpio_free(button);
    gpio_free(LED1);
    gpio_free(LED2);
    destroy_workqueue(workqueue);
    // Deallocate the finished work
    kfree(work);
}

module_init(btn_irq_init);
module_exit(btn_irq_exit);

MODULE_LICENSE("GPL");
MODULE_AUTHOR("Your Name");
MODULE_DESCRIPTION("A simple Linux driver for a GPIO interrupt");

```

```
MODULE_VERSION("0.1");
```

### 3.5 Through threaded IRQ

This method registers an interrupt that passes its work to the bottom-half using threaded IRQ.

`request_threaded_irq()` in `btn_irq_init()` registers a threaded IRQ where:

1. `IRQ_WAKE_THREAD` in `btn_irq_handler()` calls the `bottom-half()` thread.
2. The thread returns `IRQ_HANDLED` that indicates that the processing is completed successfully,

#### threadedIRQ.c

```
#include <linux/module.h>
#include <linux/gpio.h>
#include <linux/interrupt.h>
#include <linux/kernel.h>
#include <linux/workqueue.h>
#include <linux/slab.h> // kmalloc()

#define button 3 // Example GPIO number
#define LED1 4
#define LED2 5
#define ON 0
#define OFF 1

static unsigned int irq_number;
unsigned int gpio_value = OFF;

static void bottom_half(int irq, void *dev_id){
    gpio_value = gpio_get_value(LED1);
    gpio_set_value(LED1, !gpio_value);
    gpio_set_value(LED2, !gpio_value);
    // gpio_value != gpio_value;
    pr_info("Button pressed!\n");
    return IRQ_HANDLED;
}

// Interrupt handler function(top-half)
static irqreturn_t btn_irq_handler(int irq, void *dev_id)
{
    return IRQ_WAKE_THREAD;
}

static int __init btn_irq_init(void)
{
    // Request GPIOs
    int result_btn, result1, result2, result_irq;
```

```

result_btn = gpio_request(button, "sysfs");
result1 = gpio_request(LED1, "sysfs");
result2 = gpio_request(LED2, "sysfs");
if (result_btn || result1 || result2) {
    pr_info("Cannot request the GPIO\n");
    return 0;
}
// Set the direction of button GPIO
gpio_direction_input(button);
// Get the IRQ number for our GPIO
irq_number = gpio_to_irq(button);
if (irq_number < 0) {
    printk(KERN_INFO "GPIO to IRQ mapping failed\n");
    return irq_number;
}
// Request the IRQ line
result_irq = request_threaded_irq(irq_number, btn_irq_handler, bottom_half,
IRQF_TRIGGER_FALLING, "btn_irq", NULL);
if (result_irq) {
    printk(KERN_INFO "IRQ request failed\n");
    return result_irq;
}
return 0;
}

static void __exit btn_irq_exit(void) {
    free_irq(irq_number, NULL);
    gpio_free(button);
    gpio_free(LED1);
    gpio_free(LED2);
}

module_init(btn_irq_init);
module_exit(btn_irq_exit);

MODULE_LICENSE("GPL");
MODULE_AUTHOR("Your Name");
MODULE_DESCRIPTION("A simple Linux driver for a GPIO interrupt");
MODULE_VERSION("0.1");

```

## Reference

- <https://embetronicx.com/tutorials/linux/device-drivers/tasklets-dynamic-method/>
- <https://embetronicx.com/tutorials/linux/device-drivers/softirq-in-linux-kernel/>
- <https://embetronicx.com/tutorials/linux/device-drivers/workqueue-in-linux-kernel/>
- <https://embetronicx.com/tutorials/linux/device-drivers/threaded-irq-in-linux-kernel/>

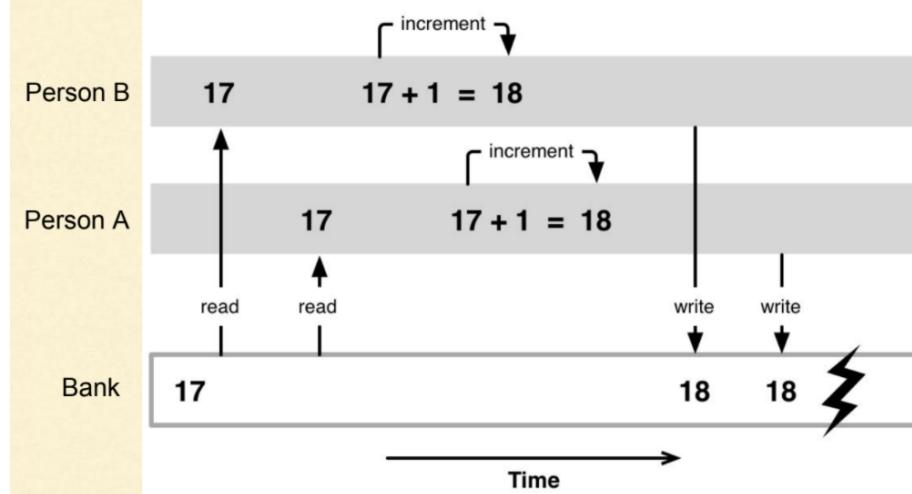
## 5. Lock

### 1. Terminology

- **Data consistency** refers to whether the same data kept at different places do or do not match.
- **Critical section** is a code segment where shared resources are accessed.
  - Example of the critical section:

```
P0
{
    Read(A);
    A=A+1;
    Write(A);
}
```

- **Race condition** is a situation where multiple processes are accessing a shared resource(**critical section**) simultaneously, which might affect **data consistency**.
  - This picture shows how the race condition occurs:

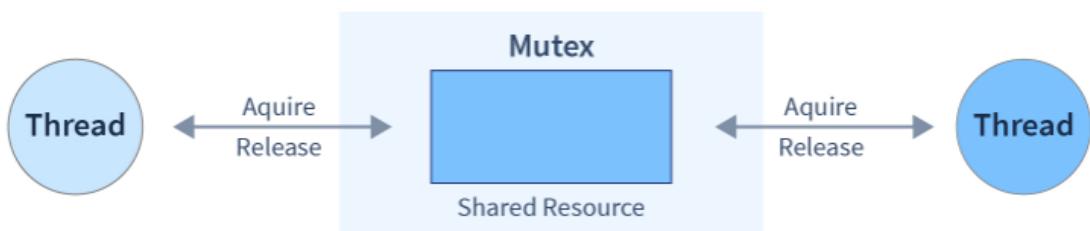


- **Process Synchronization** refers to aligning the execution timing of processes to ensure that shared resources are accessed by only one process at a time to deal with the **race condition** and maintain **data consistency**.

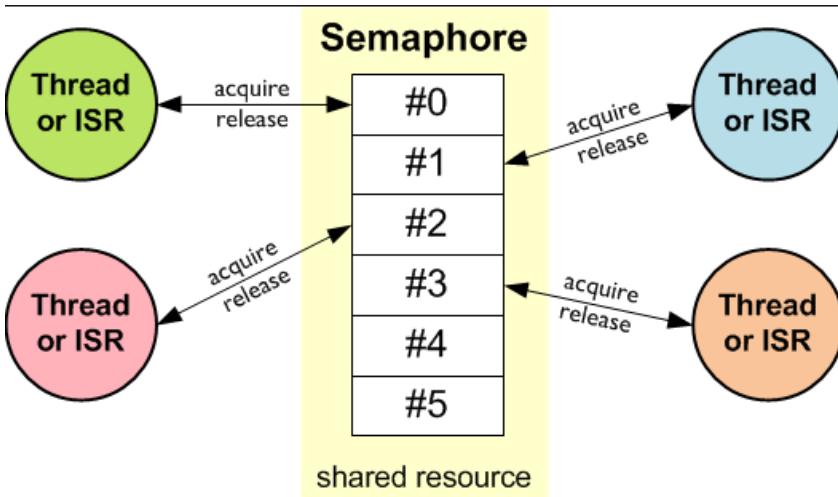
## 2. Lock mechanisms

Lock mechanisms are used to protect shared resources from concurrent access.

- **Mutex(mutual exclusion)** : Used to ensure that a shared resource is accessed only by one thread at a time.
  - It has two states: locked and unlocked. If a thread tries to lock a mutex that is already locked by another thread, the second thread waits until it is unlocked.
  - Visualization of the mutex:



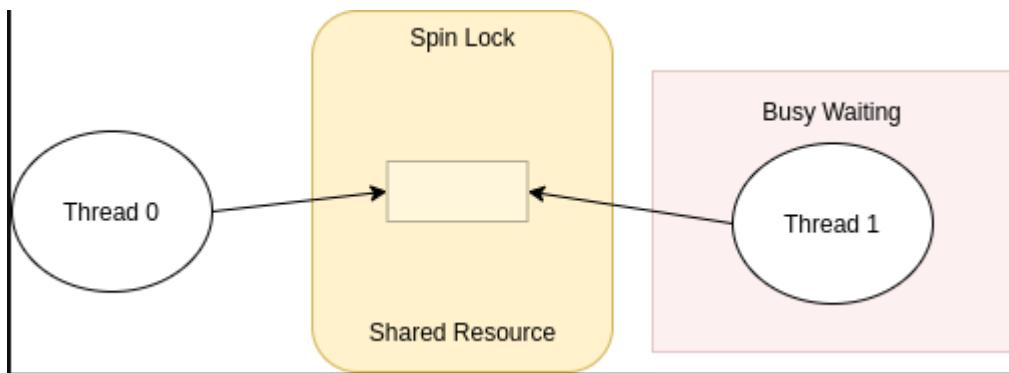
- **Semaphore** : A semaphore has multiple locks. A mutex can be considered a binary semaphore.
  - A semaphore can have multiple locks, thereby allowing more than one thread to access a shared resource but limits the number.
  - Visualization of the semaphore:



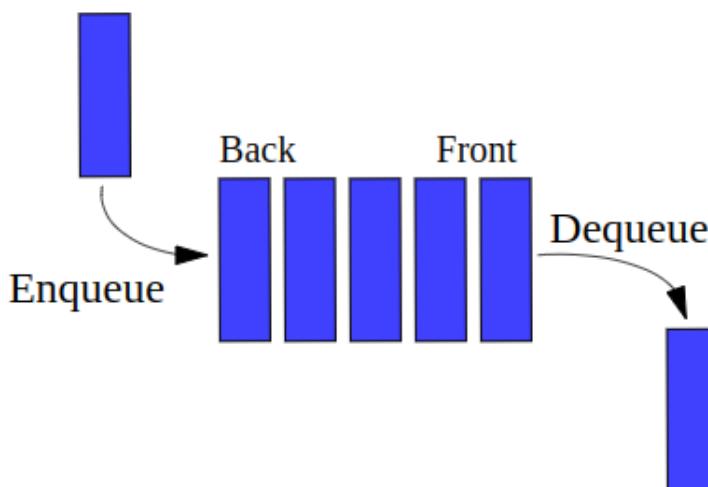
- **Monitor**: Unlike the mutex, where a thread must **explicitly** lock and unlock, monitors manage locks **implicitly**. When a thread enters a monitor's method, it automatically acquires the lock, and when it exits the method (either normally or by waiting on a condition variable), it automatically releases the lock. Monitors can use a condition variables, which are used to block a thread until a particular condition is met.

### 3. How to implement a lock

- **Spinlock(busy waiting):** When a thread attempts to acquire a spinlock that is already held by another thread, it will continuously check (or "spin") until the lock becomes available.
  - **When to use?**: This approach can be efficient if the expected wait time is short since the thread remains active and does not require a context switch.
  - **When not to use?**: However, it wastes CPU cycles if the lock is held for a long time. The **process queue** is more appropriate in this case.
  - Visualization of the spinlock:

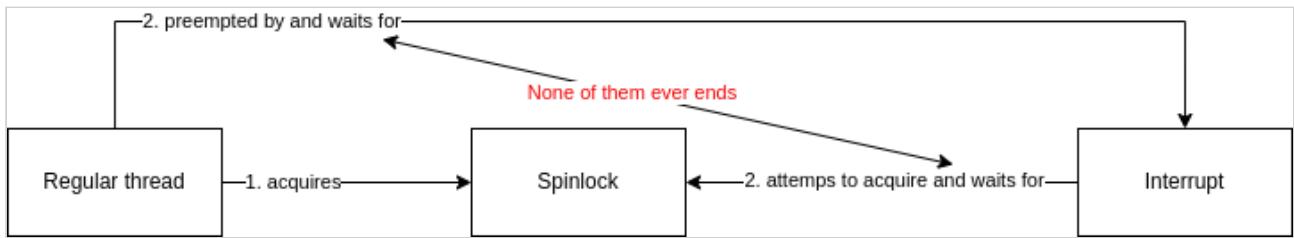


- **Process queue:** The PCB of a process is put in a queue and sent to sleep. It is popped when the shared resource becomes available.
  - Example of PCBs enqueued for sleep and dequeued to wake up for the context switch:



### 3. Lock in the interrupt context

This section describes how use of a regular spinlock leads to deadlock.



1. The regular thread acquires the spin lock to access the shared resource.
2. While the regular thread holds the spin lock, it is preempted by an interrupt.
3. The interrupt handler, which also needs to access the same shared resource, attempts to acquire the spin lock.
4. **The system is in a deadlock:** the interrupt handler cannot proceed because it's waiting for the spin lock, and the regular thread cannot proceed because it's interrupted by the handler, which does not finish.

This is why `spin_lock_irqsave()` and `spin_unlock_irqrestore()` are used for the lock in the interrupt context.

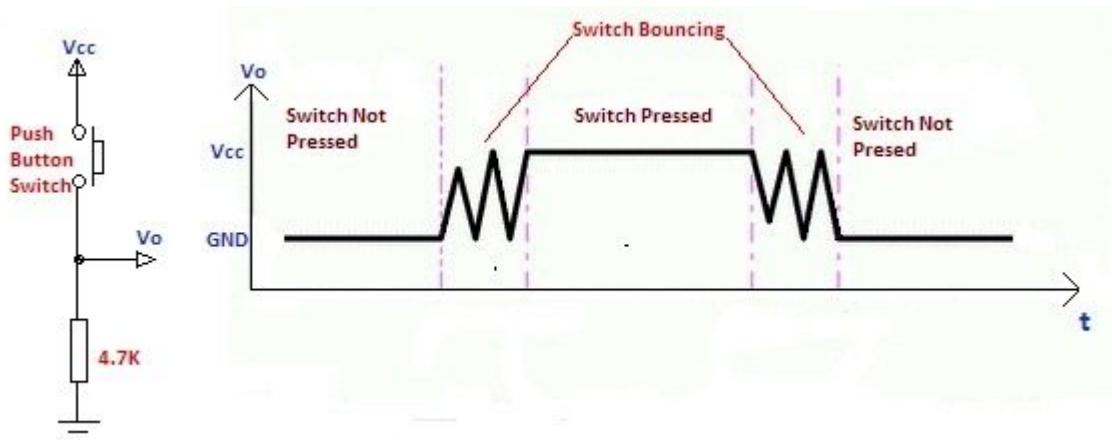
- `spin_lock_irqsave()`: disables the interrupt and saves its state
- `spin_unlock_irqrestore()`: restores the interrupt state

## 4. Practice

### 4.1 Chattering and debouncing

The hardware switch GPIO\_B4, used in the practice code in section 4.2, experiences chattering and lacks debouncing, requiring a programmatic solution.

- **Chattering**, also known as bounce, refers to the rapid, undesired opening and closing of a switch contact in a very short duration when it is pushed or released.. This phenomenon can cause multiple signals to be sent by a single actuation, leading to unpredictable behavior in electronic circuits.
- **Debouncing** is the process of removing the unwanted chattering effect from the signal generated by a mechanical switch to stabilize the input signal. This can be achieved either through software or hardware solutions.
  - Visualization of debouncing:



- In **software** debouncing, a timer can be used to ignore additional state changes that occur within a short window of time after the first change is detected.
- In **hardware** debouncing, **capacitors** can be used to filter out the noise or more complex circuits like flip-flops to stabilize the signal.
- Example of the software debouncing

```
#include <linux/delay.h>
static bool debounce_flag = false;
static void foo()
{
    unsigned int gpio_value, flags;
    if (debounce_flag)
        return IRQ_HANDLED;
    // Ignore chattering interrupts
    debounce_flag = true;
    // Enable the button after Debounce time
    mdelay(200);
    debounce_flag = false;
    // Continue code
}
```

## 4.2 Code flow

1. Register 2 button interrupts
2. Create an interrupt handler that includes a critical section where a specific value is added to the shared resource and waits for the amount of time equal to the shared resource.
3. Press the 2 buttons at the same time and check the result.

## 4.3 Example of race condition without a lock

### 4.3.1 Source code

In this example without a spinlock, The ISR triggered by the second button press is not blocked until the first button press finishes.

```
#include <linux/module.h>
#include <linux/gpio.h>
#include <linux/interrupt.h>
#include <linux/kernel.h>
#include <linux/delay.h>

#define button 3 // Example GPIO number
#define button2 36

static unsigned int irq_number;
static bool debounce_flag = false; // Used to debounce the button
static spinlock_t my_lock;
static int shared_resource = 0;

// Interrupt handler function(top-half)
static irqreturn_t btn_irq_handler(int irq, void *dev_id)
{
    return IRQ_WAKE_THREAD;
}

static irqreturn_t bottom_half(int irq, void *dev_id)
{
    unsigned int gpio_value, flags;
    if (debounce_flag)
        return IRQ_HANDLED;
    // Ignore chattering interrupts
    debounce_flag = true;
    // Enable the button after Debounce time
    mdelay(100);
    debounce_flag = false;
    // Critical section
    // spin_lock_irqsave(&my_lock, flags);
    pr_info("\nButton pressed! irq = %d, num = %d\n", irq, shared_resource);
    shared_resource += 1;
    mdelay(shared_resource * 1000);
    pr_info("Unlocked! irq = %d, num = %d\n", irq, shared_resource);
    shared_resource = 0;
    // spin_unlock_irqrestore(&my_lock, flags);
    return IRQ_HANDLED;
}

static void set_irq(int gpio, irq_handler_t bottom_half, unsigned long flag)
```

```

{
    int result_irq;
    // Get the IRQ number for our GPIO
    irq_number = gpio_to_irq(gpio);
    if (irq_number < 0) {
        printk(KERN_INFO "GPIO to IRQ mapping failed\n");
        return irq_number;
    }
    // Request the IRQ line
    result_irq = request_threaded_irq(irq_number, btn_irq_handler, bottom_half, flag,
"btn_irq", NULL);
    if (result_irq) {
        printk(KERN_INFO "IRQ request failed\n");
        return result_irq;
    }
}

static int __init btn_irq_init(void)
{
    // Request GPIO
    int result_btn;
    result_btn = gpio_request(button, "sysfs");
    if (result_btn) {
        pr_info("Cannot request the GPIO\n");
        return 0;
    }
    // Initialize the spinlock
    // spin_lock_init(&my_lock);
    // Set the direction of button GPIO
    gpio_direction_input(button);
    set_irq(button, bottom_half, IRQF_TRIGGER_FALLING);
    set_irq(button2, bottom_half, IRQF_TRIGGER_RISING);
    return 0;
}

static void __exit btn_irq_exit(void) {
    free_irq(irq_number, NULL);
    gpio_free(button);
    gpio_free(button2);
}

module_init(btn_irq_init);
module_exit(btn_irq_exit);

MODULE_LICENSE("GPL");
MODULE_AUTHOR("Your Name");
MODULE_DESCRIPTION("A simple Linux driver for a GPIO interrupt");
MODULE_VERSION("0.1");

```

### 4.3.2 Result

```
Button pressed! irq = 106, shared_resource = 0
Button pressed! irq = 107, shared_resource = 1
Unlocked! irq = 106, shared_resource = 2
Unlocked! irq = 107, shared_resource = 0
```

Simultaneous access to **shared\_resource** is not blocked, which leads to having unpredictable values.

## 4.4 Example of race condition with a lock

### 4.4.1 Source code

In this example with a spinlock, The ISR triggered by the second button press is blocked until the first button press finishes.

```
#include <linux/module.h>
#include <linux/gpio.h>
#include <linux/interrupt.h>
#include <linux/kernel.h>
#include <linux/delay.h>
#include <linux/spinlock.h>

#define button 3 // Example GPIO number
#define button2 36

static unsigned int irq_number;
static bool debounce_flag = false; // Used to debounce the button
static spinlock_t my_lock;
static int shared_resource = 0;

// Interrupt handler function(top-half)
static irqreturn_t btn_irq_handler(int irq, void *dev_id)
{
    return IRQ_WAKE_THREAD;
}

static irqreturn_t bottom_half(int irq, void *dev_id)
{
    unsigned int gpio_value, flags;
    if (debounce_flag)
        return IRQ_HANDLED;
    // Ignore chattering interrupts
    debounce_flag = true;
    // Enable the button after Debounce time
    mdelay(200);
```

```

debounce_flag = false;
// Critical section
spin_lock_irqsave(&my_lock, flags);
pr_info("\nButton pressed! irq = %d, shared_resource = %d\n", irq,
shared_resource);
shared_resource += 1;
mdelay(shared_resource * 1000);
pr_info("Unlocked! irq = %d, shared_resource = %d\n", irq, shared_resource);
shared_resource = 0;
spin_unlock_irqrestore(&my_lock, flags);
return IRQ_HANDLED;
}

static void set_irq(int gpio, irq_handler_t bottom_half, unsigned long flag)
{
    int result_irq;
    // Get the IRQ number for our GPIO
    irq_number = gpio_to_irq(gpio);
    if (irq_number < 0) {
        printk(KERN_INFO "GPIO to IRQ mapping failed\n");
        return irq_number;
    }
    // Request the IRQ line
    result_irq = request_threaded_irq(irq_number, btn_irq_handler, bottom_half, flag,
"btn_irq", NULL);
    if (result_irq) {
        printk(KERN_INFO "IRQ request failed\n");
        return result_irq;
    }
}

static int __init btn_irq_init(void)
{
    // Request GPIO
    int result_btn;
    result_btn = gpio_request(button, "sysfs");
    if (result_btn) {
        pr_info("Cannot request the GPIO\n");
        return 0;
    }
    result_btn = gpio_request(button2, "sysfs");
    if (result_btn) {
        pr_info("Cannot request the GPIO\n");
        return 0;
    }
    // Initialize the spinlock
    spin_lock_init(&my_lock);
    // Set the direction of button GPIO
    gpio_direction_input(button);
    set_irq(button, bottom_half, IRQF_TRIGGER_FALLING);
    set_irq(button2, bottom_half, IRQF_TRIGGER_RISING);
    return 0;
}

```

```

static void __exit btn_irq_exit(void) {
    free_irq(irq_number, NULL);
    gpio_free(button);
    gpio_free(button2);
}

module_init(btn_irq_init);
module_exit(btn_irq_exit);

MODULE_LICENSE("GPL");
MODULE_AUTHOR("Your Name");
MODULE_DESCRIPTION("A simple Linux driver for a GPIO interrupt");
MODULE_VERSION("0.1");

```

#### 4.4.2 Result

Simultaneous access to **shared\_resource** is not blocked, which leads to having stable values.

```

Button pressed! irq = 107, shared_resource = 0
Unlocked! irq = 107, shared_resource = 1

Button pressed! irq = 106, shared_resource = 0
Unlocked! irq = 106, shared_resource = 1

```

## 6. Memory allocation

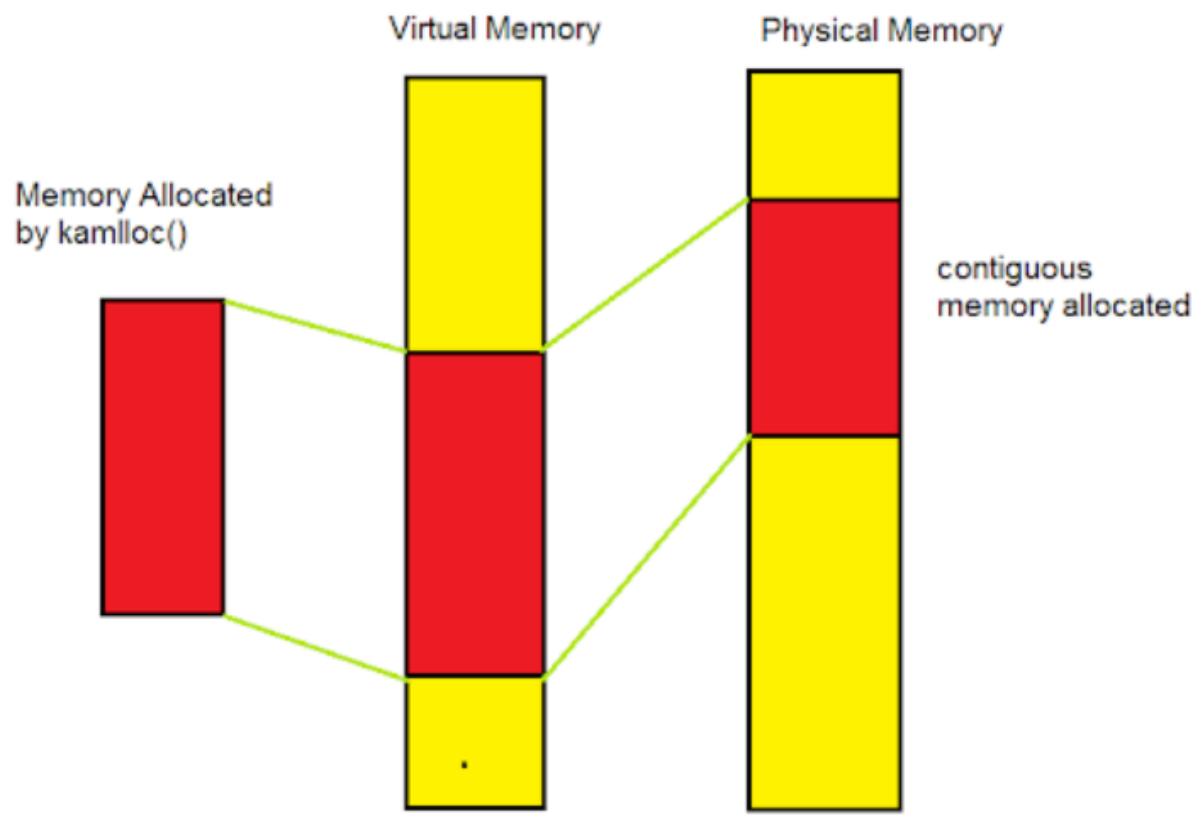
### 1. Memory allocation APIs

#### 1.1 kmalloc()

- Usage:** `kmalloc()` is used to allocate small to medium amounts of memory that are physically contiguous.
- Allocation:** `void *kmalloc(size_t size, gfp_t flags);`
  - `size` is the amount of memory you want to allocate.
  - `flags` are the GFP (Get Free Pages) flags that affect the behavior of the allocation, like blocking or non-blocking.
- Free:** `kfree(const void *ptr)` frees allocated memory

#### 1.1.1 Memory allocated by kmalloc()

The memory allocated by `kmalloc()` is contiguous in the physical memory as well as in the virtual memory.



### 1.1.2 GFP flags

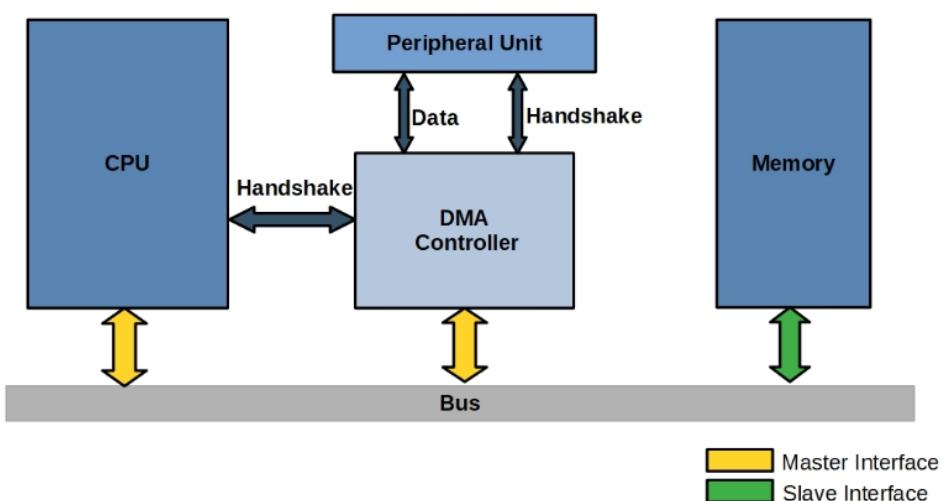
The flags used in kmalloc() are defined in `slap.h`

```
/**
 * kmalloc - allocate memory
 * @size: how many bytes of memory are required.
 * @flags: the type of memory to allocate.
 *
 * kmalloc is the normal method of allocating memory
 * for objects smaller than page size in the kernel.
 *
 * The @flags argument may be one of:
 *
 * %GFP_USER - Allocate memory on behalf of user. May sleep.
 *
 * %GFP_KERNEL - Allocate normal kernel ram. May sleep.
 *
 * %GFP_ATOMIC - Allocation will not sleep. May use emergency pools.
 *   For example, use this inside interrupt handlers.
 *
 * %GFP_HIGHUSER - Allocate pages from high memory.
 *
 * %GFP_NOIO - Do not do any I/O at all while trying to get memory.
 *
 * %GFP_NOFS - Do not make any fs calls while trying to get memory.
 *
 * %GFP_NOWAIT - Allocation will not sleep.
 *
 * %__GFP_THISNODE - Allocate node-local memory only.
 *
 * %GFP_DMA - Allocation suitable for DMA.
 *   Should only be used for kmalloc() caches. Otherwise, use a
 *   slab created with SLAB_DMA.
 *
 * Also it is possible to set different flags by OR'ing
 * in one or more of the following additional @flags:
 *
 * %__GFP_COLD - Request cache-cold pages instead of
 *   trying to return cache-warm pages.
 *
 * %__GFP_HIGH - This allocation has high priority and may use emergency pools.
 *
 * %__GFP_NOFAIL - Indicate that this allocation is in no way allowed to fail
 *   (think twice before using).
 *
 * %__GFP_NORETRY - If memory is not immediately available,
 *   then give up at once.
 *
 * %__GFP_NOWARN - If allocation fails, don't issue any warnings.
 *
 * %__GFP_REPEAT - If allocation fails initially, try once more before failing.
 *
 * There are other flags available as well, but these are not intended
 * for general use, and so are not documented here. For a full list of
 * potential flags, always refer to linux/gfp.h.
 */
```

## 1.2 vmalloc()

- Usage:** `vmalloc()` is used for allocating large amounts of memory. The memory is virtually contiguous but may not be physically contiguous, which is useful when large buffers are needed, and physical continuity is not a requirement.
- Allocation:** `void *vmalloc(unsigned long size);`
- Free:** `vfree(const void *ptr)` releases allocated memory.

## 1.3 dma-alloc()



- Usage:** Used to allocate physically contiguous memory that is DMA (Direct Memory Access) capable. There are devices that require DMA operations with contiguous physical memory.
- Allocation:** `void *dma_alloc_coherent(struct device *dev, size_t size, dma_addr_t *dma_handle, gfp_t flags)`
  - `dev` is a device structure.
  - `size` is the allocation size.
  - `dma_handle` is a pointer to a DMA address(physical address).
- Free:** `void dma_free_coherent(struct device *dev, size_t size, void *cpu_addr, dma_addr_t dma_handle)`
- Cacheable/Non-Cacheable:** `dma_alloc_coherent` typically returns non-cacheable memory to ensure data consistency, as DMA operations directly access the memory, skipping the CPU and cache.

## 1.4 Other APIs

There are several other memory allocation methods used in the Linux kernel for specific purposes.

- `get_free_pages()`
- `alloc_page()`
- `vmalloc_to_pfn()`
- High Memory Access: `kmap()`

## 2. SLUB

The SLUB is a memory allocator used in the Linux kernel, designed to be simple and efficient. It was introduced to improve upon the shortcomings of previous allocators like SLAB and SLOB, particularly on multicore systems.

### Features and Advantages

- **Efficiency:** SLUB minimizes the use of locks and reduces fragmentation, making it more efficient, especially in SMP (Symmetric Multiprocessing) environments.
- **Scalability:** It scales well with the number of CPUs, making it suitable for high-performance and high-throughput systems.
- **Simplicity:** Its design is simpler than that of SLAB, making it easier to maintain and understand.
- **Debugging Support:** SLUB comes with extensive debugging support to detect common errors such as double frees, memory overruns, and usage of freed objects.

### How It Works

- **Caches and Objects:** SLUB operates by managing caches of objects, where each cache is tailored to a specific size of the object. This helps in optimizing memory usage and access speed.
- **Allocation:** When a request for memory allocation is made, SLUB tries to satisfy the request from the corresponding object cache. If there is a free object available, it is returned to the caller.
- **Freeing Memory:** When memory is freed, the object is returned to its cache, making it available for future allocations.
- **Per-CPU Caches:** To improve performance on multicore systems, SLUB maintains per-CPU caches, which reduce the need for locking and increase the speed of allocation and deallocation operations.

## 3. Practice

### Code explanation

- Memory allocation and deallocation are executed on each button press by turns.
- `platform_driver` is used to map the DMA allocation to a (dummy)device that requires DMA operations.
- Each memory allocation API returns `NULL` if it failed.

`bottom_half()` is implemented in each practice section.

```
#include <linux/module.h>
#include <linux/platform_device.h>
```

```

#include <linux/gpio.h>
#include <linux/interrupt.h>

#include <linux/delay.h>
#include <linux/slab.h> // For kmalloc() and kfree()
#include <linux/vmalloc.h> // For vmalloc() and vfree()
#include <linux/dma-mapping.h> // For dma_alloc_coherent() and dma_free_coherent()
#include <linux/delay.h> // For mdelay()

#define button 3 // Example GPIO number

static struct device *dev;
static unsigned int irq_number;
static int toggle = 0;
static void *buffer;

// Interrupt handler function(top-half)
static irqreturn_t btn_irq_handler(int irq, void *dev_id)
{
    return IRQ_WAKE_THREAD;
}

static irqreturn_t bottom_half(int irq, void *dev_id);

static int set_irq(int gpio, irq_handler_t bottom_half, unsigned long flag)
{
    int result_irq;
    // Get the IRQ number for our GPIO
    irq_number = gpio_to_irq(gpio);
    if (irq_number < 0) {
        printk(KERN_INFO "GPIO to IRQ mapping failed\n");
        return 1;
    }
    // Request the IRQ line
    result_irq = request_threaded_irq(irq_number, btn_irq_handler, bottom_half, flag,
"btn_irq", NULL);
    if (result_irq) {
        printk(KERN_INFO "IRQ request failed\n");
        return 1;
    }
    return 0;
}

static int my_platform_probe(struct platform_device *pdev)
{
    pr_info("my_platform_probe\n");
    dev = &pdev->dev;
    return 0;
}

static int my_platform_remove(struct platform_device *pdev)
{
}

```

```

    dev_info(&pdev->dev, "Platform device removed\n");
    return 0;
}

static struct of_device_id testdev_of_match[] = {
    { .compatible = "yang,mydevice", },
    {},
};

MODULE_DEVICE_TABLE(of, testdev_of_match);

static struct platform_driver my_platform_driver = {
    .probe = my_platform_probe,
    .remove = my_platform_remove,
    .driver = {
        .name = "my_platform_driver",
        .of_match_table = testdev_of_match,
        .owner = THIS_MODULE,
    },
};

// Module initialization
static int __init testdev_init(void)
{
    platform_driver_register(&my_platform_driver);
    // Request GPIO
    int result_btn;
    result_btn = gpio_request(button, "sysfs");
    if (result_btn) {
        pr_info("Cannot request the LED GPIO\n");
        return 1;
    }
    set_irq(button, bottom_half, IRQF_TRIGGER_FALLING);
    return 0;
}

// Module exit
static void __exit testdev_exit(void)
{
    platform_driver_unregister(&my_platform_driver);
    free_irq(irq_number, NULL);
    gpio_free(button);
}

module_init(testdev_init);
module_exit(testdev_exit);

MODULE_LICENSE("GPL");
MODULE_AUTHOR("Your Name");
MODULE_DESCRIPTION("Practice module");

```

## 3.1 kmalloc()

### 3.1.1 Source code

```
static irqreturn_t bottom_half(int irq, void *dev_id)
{
    if (!toggle) {
        // Allocate memory using kmalloc
        buffer = kmalloc(sizeof(char), GFP_ATOMIC);
        if (!buffer)
            return -ENOMEM;
        printk(KERN_INFO "kmalloc allocated, Address: %p\n", (char *)buffer);
    } else {
        // Free memory
        kfree(buffer);
        printk(KERN_INFO "kmalloc freed, Address: %p\n");
    }
    toggle = !toggle;
    return IRQ_HANDLED;
}
```

### 3.1.2 Result

```
root@s5p6818:~# [ 4128.076000] kmalloc allocated, Address: ffffffc00a3a2700
[ 4128.272000] kmalloc freed, Address: 0000000000000001
```

## 4.2 vmalloc()

### 4.2.1 Source code

```
static irqreturn_t bottom_half(int irq, void *dev_id)
{
    if (!toggle) {
        // Allocate memory using vmalloc
        buffer = vmalloc(sizeof(char));
        if (!buffer)
            return -ENOMEM; // Return error if allocation fails
        printk(KERN_INFO "vmalloc allocated, Address: %p\n", (char *)buffer);
    } else {
        // Free memory
        vfree(buffer);
```

```

        printk(KERN_INFO "vmalloc freed, Address: %p\n", (char *)buffer);
    }
    toggle = !toggle;
    return IRQ_HANDLED;
}

```

#### 4.2.2 Result

```

cat /proc/vmallocinfo | grep bottom_half

root@s5p6818:~# [ 4248.104000] vmalloc allocated, Address: ffffff80090b6000
root@s5p6818:~# cat /proc/vmallocinfo | grep bottom_half
0xfffffff80090b6000-0xffffffff80090b8000      8192 bottom_half+0x20/0x88 [interrupt] pages=1 vmalloc

```

### 4.3 dma\_alloc\_coherent()

#### 4.3.1 Source code

```

static irqreturn_t bottom_half(int irq, void *dev_id)
{
    dma_addr_t dma_handle;
    size_t size = 1024;
    // Allocate memory using dma_alloc_coherent
    if (!toggle) {
        buffer = dma_alloc_coherent(dev, size, &dma_handle, GFP_ATOMIC);
        if (!buffer) {
            printk(KERN_INFO "dma_alloc failed\n");
            return -ENOMEM; // Return error if allocation fails
        }
        printk(KERN_INFO "dma_alloc allocated, Address: %p", dma_handle);
    } else {
        // Free memory
        dma_free_coherent(dev, size, buffer, dma_handle);
        printk(KERN_INFO "dma_alloc freed, Address: %p", dma_handle);
    }
    toggle = !toggle;
    return IRQ_HANDLED;
}

```

#### 4.3.2 Result

```

root@s5p6818:~# [ 4417.744000] dma_alloc allocated, Address: 000000004c800000
[ 4417.984000] dma_alloc freed, Address: ffffff80080e127c

```

## 7. Timer

### 1. Jiffies

The kernel uses jiffies to measure time intervals when low resolution is tolerable.

A jiffy is the duration between two consecutive ticks of the system timer.

#### 1.1 Uses

- **Timekeeping:** They are used for time-related functions within the kernel. For example, The scheduler uses jiffies to decide when to switch between processes, allowing the kernel to allocate CPU time effectively among running processes.
- **Timer Management:** Timers and time-based delays (like `msleep()`) are implemented using jiffies, allowing the kernel and kernel modules to perform actions after a specified number of jiffies has passed.

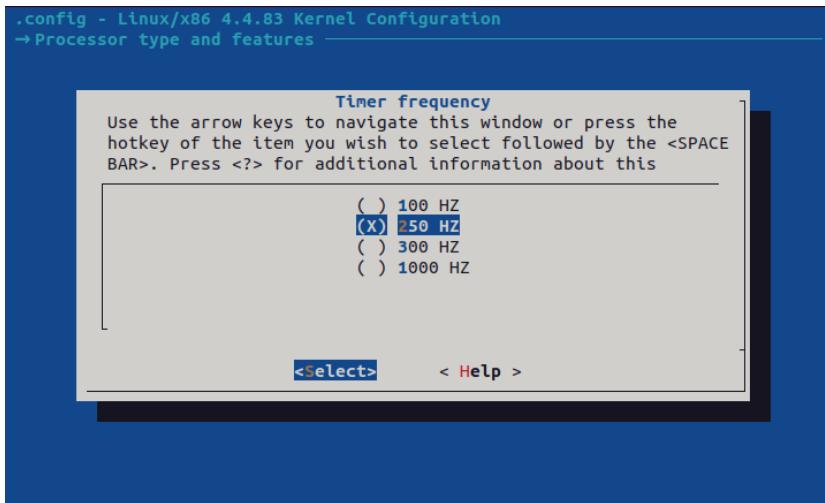
#### 1.2 Limitation

- **Resolution:** For tasks requiring higher time resolution than jiffies can provide, the kernel uses other mechanisms, such as high-resolution timers (hrtimers), which can offer nanosecond precision.

#### 1.3 Resolution change

The timer frequency can be changed can be changed as follows

1. make menuconfig
2. Processor type and features
3. Timer frequency



## 2. Hrtimer

### 2.1 Advantages over traditional timers

- **Precision:** The main advantage of hrtimers is their ability to provide precise timing, crucial for applications where timing accuracy is paramount. Hrtimers can provide timing resolutions in the nanosecond range, far exceeding the granularity offered by jiffies.
- **Dynamic:** They adjust dynamically to the resolution supported by the underlying hardware, making the most of available timer hardware capabilities.
- **Kernel Integration:** Hrtimers are fully integrated with the kernel's timekeeping infrastructure, allowing them to be used for a wide range of high-precision timing tasks.

### 2.2 Uses

Hrtimers are suitable for tasks that require high precision.

- **Networking:** High-resolution timers are crucial in networking for tasks such as managing packet transmission intervals, timeout handling, and implementing precise time-sensitive protocols.
- **Multimedia:** For multimedia applications, precise timing is essential for audio and video synchronization, playback control, and real-time streaming.
- **Real-Time Applications:** Real-time systems that require deterministic behavior and precise timing, such as industrial control systems, robotics, and real-time simulations, rely heavily on hrtimers.

## 3. getnstimeofday

`getnstimeofday()` retrieves the current time with nanosecond precision. It fills a data structure with the current time, broken down into seconds and nanoseconds.

## 3.1 Overview

- **Purpose:** The primary goal of `getnstimeofday()` is to obtain the current time with high precision. It is used in scenarios where detailed time information is necessary, surpassing the granularity that seconds or milliseconds can offer.
- **Return Type:** It populates a `timespec` (or `timespec64` in newer kernels) structure, which consists of two fields:
  - `tv_sec` : Represents the number of seconds elapsed since the Unix epoch (00:00:00 UTC on 1 January 1970).
  - `tv_nsec` : Represents the number of nanoseconds past the second. The value is in the range of 0 to 999,999,999 nanoseconds.

## 4. Practice

### 4.1 Jiffies

#### 4.1.1 Source code

- **Access to Jiffies:** Jiffies are stored as a global variable (`unsigned long int`) that is incremented by the system timer interrupt at each timer tick. The current value of jiffies can be accessed directly to measure elapsed time or schedule future actions.
- **HZ:** The frequency at which jiffies are incremented is defined by the `HZ` value in the kernel. This value represents the number of timer ticks per second and varies across different hardware platforms and kernel configurations. Higher `HZ` values offer finer-grained time resolution but can increase overhead due to more frequent timer interrupts.
  - e.g. 250 HZ = trigger per 1/250 second = resolution of 4 ms

This function prints a message when 1 second has elapsed.

```
#include <linux/module.h> // Needed by all modules
#include <linux/kernel.h> // Needed for KERN_INFO
#include <linux/init.h> // Needed for the macros
#include <linux/jiffies.h> // Needed for jiffies
#include <linux/gpio.h>
#include <linux/interrupt.h>

#define button 3 // Example GPIO number

static unsigned int irq_number;

// Interrupt handler function(top-half)
static irqreturn_t btn_irq_handler(int irq, void *dev_id)
```

```

{
    return IRQ_WAKE_THREAD;
}

static irqreturn_t bottom_half(int irq, void *dev_id)
{
    unsigned long jiffies_at_start = jiffies; // Capture the current value of jiffies
    unsigned long delay = 1 * HZ; // Delay of 1 second; HZ is the number of jiffies
    per second
    // Busy wait until the specified delay has elapsed
    while (time_before(jiffies, jiffies_at_start + delay))
    {
        // This loop will keep running until the delay has passed
        // In a real application, you generally don't want to use busy waiting!
        // This is just for demonstration.
    }
    printk(KERN_INFO "Jiffies: 1 second has elapsed.\n");
    return IRQ_HANDLED;
}

static int set_irq(int gpio, irq_handler_t bottom_half, unsigned long flag)
{
    int result_irq;
    // Get the IRQ number for our GPIO
    irq_number = gpio_to_irq(gpio);
    if (irq_number < 0) {
        printk(KERN_INFO "GPIO to IRQ mapping failed\n");
        return 1;
    }
    // Request the IRQ line
    result_irq = request_threaded_irq(irq_number, btn_irq_handler, bottom_half, flag,
"btn_irq", NULL);
    if (result_irq) {
        printk(KERN_INFO "IRQ request failed\n");
        return 1;
    }
    return 0;
}

static int __init testdev_init(void)
{
    set_irq(button, bottom_half, IRQF_TRIGGER_FALLING);
    return 0;
}

// Module exit
static void __exit testdev_exit(void)
{
    free_irq(irq_number, NULL);
}

module_init(testdev_init);
module_exit(testdev_exit);

```

```
MODULE_LICENSE("GPL");
MODULE_AUTHOR("Your Name");
MODULE_DESCRIPTION("Practice module");
```

#### 4.1.2 Result

```
Jiffies: 1 second has elapsed.
```

### 4.2 Getnstimeofday

`getnstimeofday()` has been deprecated and replaced with `timespec64` and `ktime_get_real_ts64()` to solve the year 2038 problem.

#### 4.2.1 Source code

Same code as above except `bottom_half()`

```
static irqreturn_t bottom_half(int irq, void *dev_id)
{
    struct timespec now;
    // Get current time
    getnstimeofday(&now);
    // Print current time
    printk(KERN_INFO "Current time: %ld seconds and %ld nanoseconds\n", now.tv_sec,
    now.tv_nsec);
    return IRQ_HANDLED;
}
```

#### 4.2.2 Result

```
Current time: 1600602783 seconds and 493494300 nanoseconds
```

### 4.3 Hrtimer

#### 4.3.1 Source code

`timer_callback()` is called when the hrtimer has fired

```
#include <linux/module.h> // Needed by all modules
#include <linux/kernel.h> // Needed for KERN_INFO
#include <linux/init.h> // Needed for the macros
```

```

#include <linux/gpio.h>
#include <linux/interrupt.h>
#include <linux/timekeeping.h>

#define button 3 // Example GPIO number

static unsigned int irq_number;
static struct hrtimer my_timer;
static ktime_t timer_period, start_time;

// Interrupt handler function(top-half)
static irqreturn_t btn_irq_handler(int irq, void *dev_id)
{
    return IRQ_WAKE_THREAD;
}

enum hrtimer_restart timer_callback(struct hrtimer *timer)
{
    ktime_t elapsed;
    elapsed = ktime_sub(ktime_get(), start_time); // Calculate elapsed time
    printk(KERN_INFO "Timer fired. Requested duration: %lld ns, Actual duration: %lld
ns\n",
          timer_period, ktime_to_ns(elapsed));
    return HRTIMER_NORESTART;
}

static irqreturn_t bottom_half(int irq, void *dev_id)
{
    // 1ms = 1,000,000ns
    timer_period = ktime_set(0, 100 * 1000000); // 100ms in ns
    // Initialize the high-resolution timer
    hrtimer_init(&my_timer, CLOCK_MONOTONIC, HRTIMER_MODE_REL);
    my_timer.function = timer_callback;
    start_time = ktime_get(); // Capture start time
    hrtimer_start(&my_timer, timer_period, HRTIMER_MODE_REL);
    return IRQ_HANDLED;
}

static int set_irq(int gpio, irq_handler_t bottom_half, unsigned long flag)
{
    int result_irq;
    // Get the IRQ number for our GPIO
    irq_number = gpio_to_irq(gpio);
    if (irq_number < 0) {
        printk(KERN_INFO "GPIO to IRQ mapping failed\n");
        return 1;
    }
    // Request the IRQ line
    result_irq = request_threaded_irq(irq_number, btn_irq_handler, bottom_half, flag,
                                     "btn_irq", NULL);
    if (result_irq) {
        printk(KERN_INFO "IRQ request failed\n");
        return 1;
    }
}

```

```

    }
    return 0;
}

static int __init testdev_init(void)
{
    set_irq(button, bottom_half, IRQF_TRIGGER_FALLING);
    return 0;
}

// Module exit
static void __exit testdev_exit(void)
{
    int ret;
    free_irq(irq_number, NULL);
    ret = hrtimer_cancel(&my_timer);
    if (ret) {
        printk(KERN_INFO "The timer was still in use.\n");
    }
}

module_init(testdev_init);
module_exit(testdev_exit);

MODULE_LICENSE("GPL");
MODULE_AUTHOR("Your Name");
MODULE_DESCRIPTION("Practice module");

```

#### 4.3.2 Result

**hrtimer** is functional but is expected to support nano second precision, which the target board does not support.

```
Timer fired. Requested duration: 100000000 ns, Actual duration: 100104700 ns
```

#### Reference

- <https://embetronicx.com/tutorials/linux/device-drivers/using-kernel-timer-in-linux-device-driver/>
- <https://dataonair.or.kr/db-tech-reference/d-lounge/technical-data/?mod=document&uid=236817>

# Memo

```
# Find files containing a specific string
grep -rl 'string1' .

# Find files containing specific strings
grep -rl 'string1' ./ | xargs grep -l 'string2'

# Find files with a specific filename
find ./ -type f -name "filename"

# Find files with a specific filename and a specific string
find ./ -type f -name "filename" -exec grep -l "string" {} +
```

- Module Driver로 제작
  - insmod / rmmod 지원할 것.
- GPIO 관련된 모든 내용은 register를 직접 Access 하여 구현하시오.
- misc driver로 제작
- DTS를 통해 2개 이상의 device를 생성하시오.
  - 그중 하나의 device는 hrtimer에서 on/off 용으로 사용.
  - 다른 device는 Input/Output 용으로 ioctl을 테스트하도록 하시오.
- System call : open / close / read / write / ioctl
  - 각 systemcall 호출 시 print 출력
- Gpio button interrupt
  - DTS에 gpio 지정
  - Probe 시 interrupt thread 등록
  - Button press/release 시 interrupt thread에서 print 출력
  - spin lock 사용.
- Timer
  - probe 시 hrtimer 등록
  - 특정 시간마다 등록된 특정 device 가 timer에 맞게 on/off 되도록 하시오.
    - ioctl을 통해 시간 설정을 할 수 있게 하시오.

# U-boot

U-Boot is an open-source boot loader used in embedded devices to perform various low-level hardware initialization tasks and boot the device's operating system kernel.

## 1. Device Driver

### 1

#### Driver Model

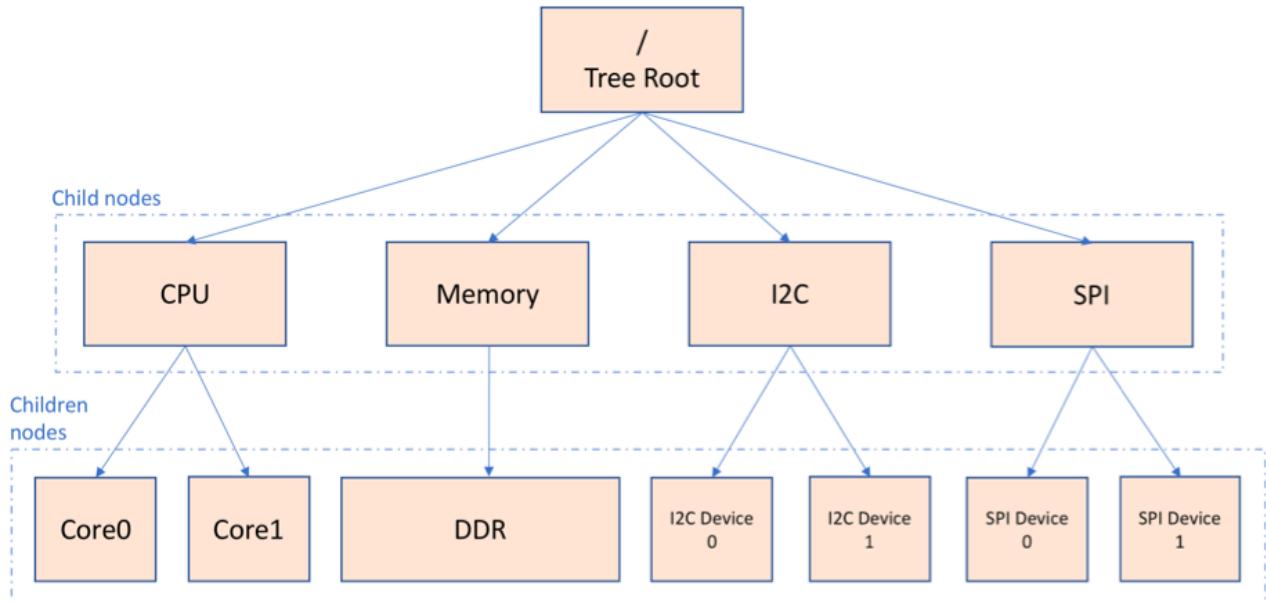
- **Standardization:** The driver model in U-Boot aims to standardize the way in which hardware drivers are written and integrated. Each driver under this model has a standard set of functions. This helps in maintaining consistency across different hardware platforms.

#### **UCLASS\_DRIVER and UCLASS\_DEVICE**

These are 2 classes within the U-Boot Driver Model:

- **UCLASS\_DRIVER:**
  - This is an abstraction class of drivers that allows U-Boot to handle drivers of a similar kind in a uniform way, irrespective of the underlying hardware specifics.
  - Each driver class typically corresponds to a certain type of hardware functionality, like serial I/O, disk interfaces, etc.
- **UCLASS\_DEVICE:**
  - Refers to the actual devices that are instantiated based on the drivers(UCLASS\_DRIVER).
  - UCLASS\_DEVICE allows for the actual hardware-specific operations to be performed. It's where the physical device is managed, and it interacts with the UCLASS\_DRIVER to ensure the correct functioning of the hardware.
  - Each device under UCLASS\_DEVICE will be initialized according to the specifications of its corresponding UCLASS\_DRIVER.

## Device Tree



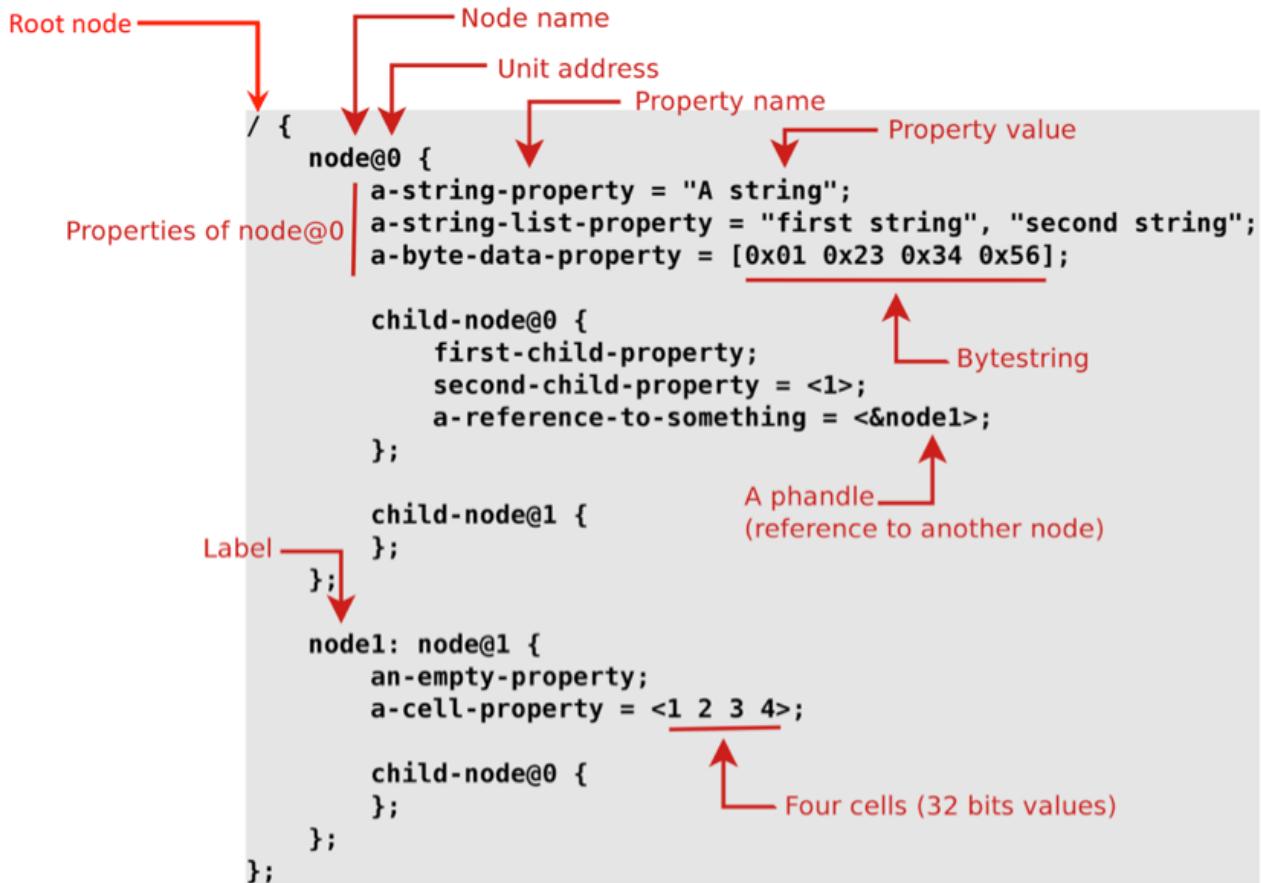
### What is Device Tree?

- **Hardware Description:** The Device Tree is a tree data structure where each node represents a hardware component in a system.
- **Formats:** The Device Tree can exist in two formats:
  - **Device Tree Source (DTS):** This is a human-readable and editable text file that describes the hardware.
  - **Device Tree Blob (DTB):** This is a binary file compiled from the DTS, used by the operating system or bootloader.

### Why Use Device Tree?

- **Hardware Abstraction:** The Device Tree provides a way to abstract the hardware details. It allows the software to adapt to different hardware without needing changes in the code. It enables the addition of new hardware configurations without requiring modifications in the kernel source code.
- **Bootloader and Kernel Simplification:** The Device Tree simplifies the bootloader and kernel code by removing the need for hard-coded hardware information. This makes the code cleaner and more maintainable.
- **Device configuration parameters:** The Device Tree includes information about device parameters, interrupt lines, memory addresses, etc., which are crucial for the correct operation of the hardware.

## Device tree structure



## FDT

### What is Flattened Device Tree (FDT)?

- Purpose of FDT:** The Flattened Device Tree, also called Device Tree Blob (DTB), is a compact binary representation of the device tree. It's designed to be easily parsed by software, particularly by the bootloader, and the kernel.
- Structure:** The FDT is essentially a serialized version of the device tree in a format that is simpler for software to process.

### Using `fdt_get`, `fdt_set`, `fdt_list`

These commands are used in a bootloader or operating system that needs to interact with the FDT

## Initialization

**fdt ddr [-c] <addr>**: Set the fdt location to <addr>

```
bitminer# fdt addr -c  
The address of the fdt is 0x4daa1800  
bitminer# fdt addr 0x4daa1800
```

**fdt list**

This command is used to list the contents of the FDT.

```
bitminer# fdt list
/ {
    #address-cells = <0x00000001>;
    #size-cells = <0x00000001>;
    model = "Bitminer board based on Nexell s5p6818";
    cpu-model = "S5p6818";
    compatible = "nexell,bitminer", "nexell,s5p6818";
    chosen {
    };
    aliases {
    };
    memory {
    };
    mmc@c0069000 {
    };
    mmc@c0068000 {
    };
    mmc@c0062000 {
    };
    ethernet@c0060000 {
    };
    i2c@c00a4000 {
    };
    i2c@c00a5000 {
    };
    i2c@c00a6000 {
    };
    dp@c0102800 {
    };
    dp@c0102c00 {
    };
    usbhost@c0030000 {
    };
    dwc2otg@c0040000 {
    };
    gpio@c001a000 {
    };
    gpio@c001b000 {
    };
    gpio@c001c000 {
    };
    gpio@c001d000 {
    };
    gpio@c001e000 {
    };
    gpio@c0010800 {
    };
    pinctrl@c0010000 {
    };
    i2c_gpio@0 {
    };
    voltage-regulators {
    };
};
```

**fdt\_set**

`fdt_set` is used to modify the FDT. This could involve changing a hardware configuration parameter before the kernel is booted.

```
fdt set / model "hi hello"
bitminer# fdt list
/ {
    #address-cells = <0x00000001>;
    #size-cells = <0x00000001>;
    model = "hi hello";
    ...
}
```

**fdt\_get**

The `fdt get value` command is used to retrieve a property value from a node in the FDT and store it in a U-Boot environment variable.

```
bitminer# fdt get value my_model / model
bitminer# print my_model
my_model=hi hello
```

**menuconfig**

menuconfig is used to configure options of Linux, U-boot, etc

**Install dependencies**

```
sudo apt-get install libncurses5-dev
```

**Run menuconfig**

```
cd [workspace]/dunfell-bitminer/sources/boot/u-boot/u-boot-2016.01
make menuconfig
```

**Exclude the i2c device driver**

This is done to test how to modify u-boot through menuconfig

youngsik@youngsik-HP-Elite-Tower-800-G9-Desktop-PC: ~/w...

.config - U-Boot 2016.01 Configuration

U-Boot 2016.01 Configuration

Arrow keys navigate the menu. <Enter> selects submenus ---> (or empty submenus ----). Highlighted letters are hotkeys. Pressing <Y> includes, <N> excludes, <M> modularizes features. Press <Esc><Esc> to exit, <?> for Help, </> for Search. Legend: [\*] built-in [ ]

↑(–) ARM architecture --->  
General setup --->  
Boot images --->  
Android support commands --->  
Command line interface --->  
[ ] Console recording  
Device Tree Control --->  
[\*] Networking support --->  
Device Drivers ---> **File systems ----**  
↓(+)

<Select> < Exit > < Help > < Save > < Load >

The screenshot shows a terminal window titled "youngsik@youngsik-HP-Elite-Tower-800-G9-Desktop-PC: ~/w...". The window displays the ".config - U-Boot 2016.01 Configuration" menu, specifically the "Device Drivers → I2C support" section. A help message at the top explains the menu navigation using arrow keys, enter for selection, and various hotkeys like Y, N, M, Esc, and ?.

**I2C support**

Arrow keys navigate the menu. <Enter> selects submenus ---> (or empty submenus ----). Highlighted letters are hotkeys. Pressing <Y> includes, <N> excludes, <M> modularizes features. Press <Esc><Esc> to exit, <?> for Help, </> for Search. Legend: [\*] built-in [ ]

The menu lists the following options:

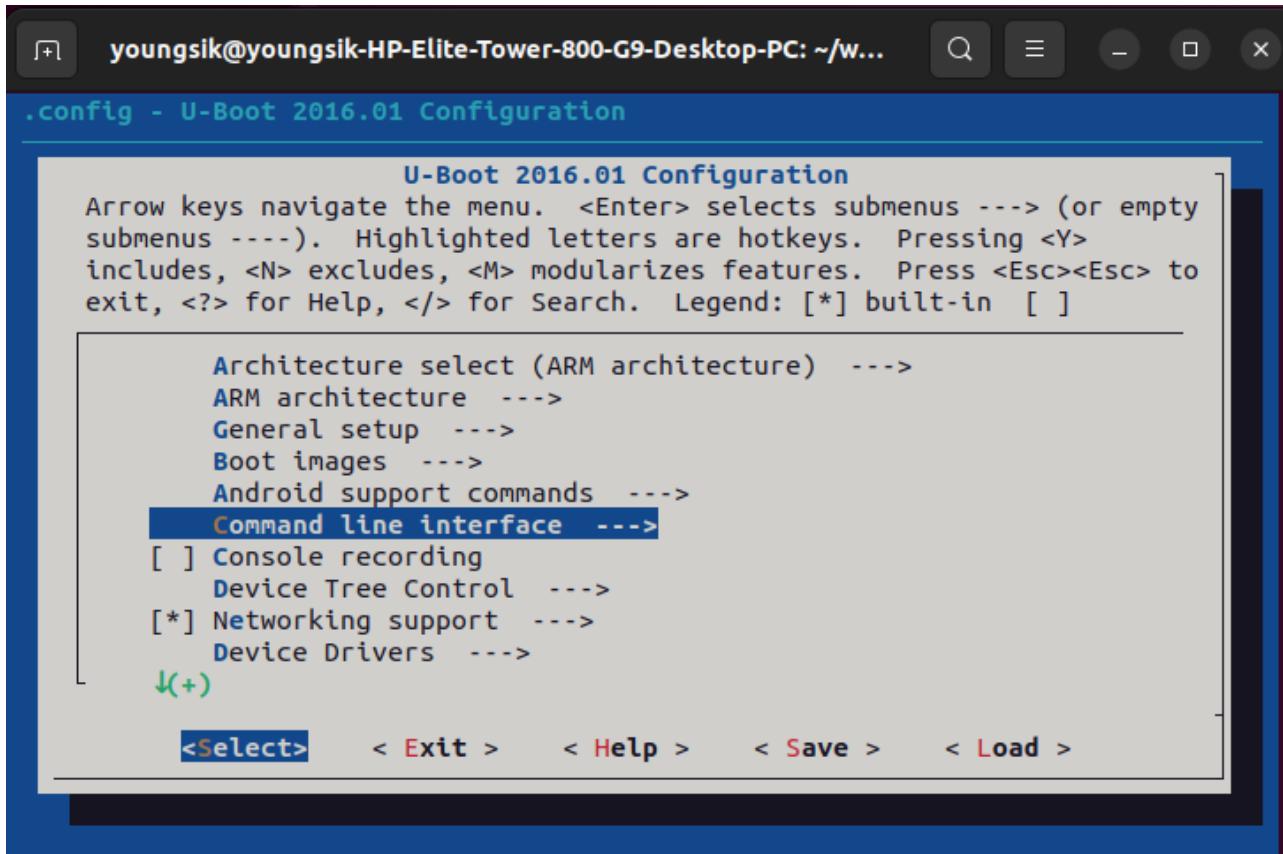
- [\*] Enable Driver Model for I2C drivers
- [ ] Enable I2C compatibility layer
- [\*] Enable Driver Model for software emulated I2C bus driver
- [ ] Rockchip I2C driver
- [\*] Nexell I2C driver
- [ ] Support I2C multiplexers

At the bottom of the menu, there are navigation keys: <Select>, < Exit >, < Help >, < Save >, and < Load >.

This results in an error in the build.

## Exclude the i2c command

Uncheck the command line in menuconfig



The screenshot shows a terminal window titled ".config - U-Boot 2016.01 Configuration". The window contains the U-Boot configuration menu. The "Command line interface" option is selected, indicated by a blue border around its submenu. The submenu includes options like "Console recording", "Device Tree Control", "Networking support", and "Device Drivers". The "Networking support" option is currently checked, indicated by a blue asterisk (\*) before the option name. At the bottom of the menu, there are navigation keys: <Select>, < Exit >, < Help >, < Save >, and < Load >. The background of the terminal window is dark blue.

```
.config - U-Boot 2016.01 Configuration
→ Command line interface → Device access commands
    Device access commands
Arrow keys navigate the menu. <Enter> selects submenus ---> (or empty
submenus ----). Highlighted letters are hotkeys. Pressing <Y>
includes, <N> excludes, <M> modularizes features. Press <Esc><Esc> to
exit, <?> for Help, </> for Search. Legend: [*] built-in [ ]
```

- [\*] dm - Access to driver model information
- [ ] demo - Demonstration commands for driver model
- [\*] loadb
- [\*] loads
- [ ] flinfo, erase, protect
- [ ] nand
- [ ] sf
- [ ] sspi
- [\*] i2c**
- [\*] usb

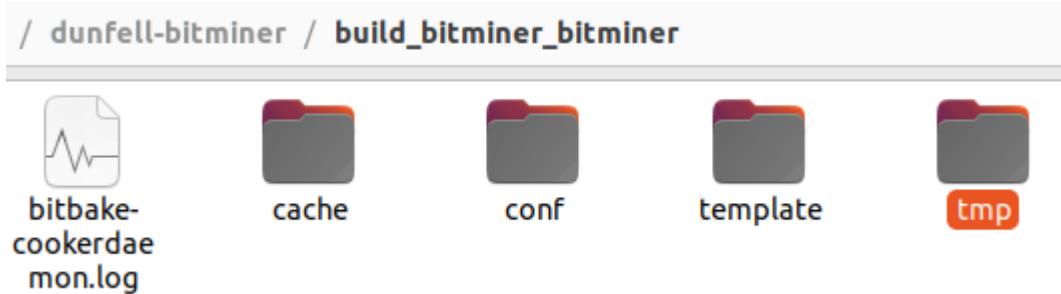
<Select>    < Exit >    < Help >    < Save >    < Load >

Make a defconfig file and copy it to the configs directory.

```
make savedefconfig
cp defconfig ./configs/s5p6818_bitminer_defconfig
```

The i2c command is excluded in the build.

Delete the `tmp` folder for clean build before building the modified u-boot



## Rebuild u-boot

```
bitbake u-boot-nexell
bitbake boot-binary
bitbake optee-build
```

### Check if the i2c command was successfully excluded

```
U-BootDRAM: 219 MiB
POFFHIS: NONE
PONHIS : PWRONPON
CHGS : CHG OFF
DCDC1 : 1250mV, En, dclim:1303026384, dclimsden:0
DCDC2 : 1200mV, En, dclim:1303026384, dclimsden:0
DCDC3 : 3300mV, En, dclim:1303026384, dclimsden:0
DCDC4 : 1500mV, En, dclim:1303026384, dclimsden:0
DCDC5 : 1500mV, En, dclim:1303026384, dclimsden:0
MMC: NEXELL DWMMC: 0
In: serial
Out: serial
Err: serial
Net:
Warning: ethernet@c0060000 (eth0) using random MAC address - 86:e2:d9:ef:ec:60
eth0: ethernet@c0060000
Hit any key to stop autoboot: 0
bitminer#
bitminer#
bitminer#
bitminer#
bitminer#
bitminer# i2c
Unknown command 'i2c' - try 'help'
bitminer#
```

## 2. U\_BOOT\_CMD

### How to add and delete a custom command in u-boot

#### Write code for the custom command

```
path/to/u-boot/common/hello.c
```

```
#include <common.h>
#include <command.h>

static int do_print_hello(cmd_tbl_t * cmdtp, int flag, int argc,
                          char * const argv[]) {
    printf("Yang Hello world!\n");
    return 0;
}

U_BOOT_CMD(
    hello,           // Command name
    1,               // Max number of arguments
    0,               // Repeatable
    do_print_hello, // Command function
    "print \"Yang Hello world!(C)\"", // Command description
    "Usage: hello" // Help text
);
```

## Add to the compile list

common/Makefile

```
obj-$(CONFIG_CMD_HELLO) += hello.o
```

This mean "Include the `hello.o` in the build based on the configuration option `CONFIG_CMD_HELLO`"

common/kconfig

```
menu "My commands"

config CMD_HELLO
    bool "Enable 'hello' command"
    default y
    help
        print "Hello world!(Kconfig)"
endmenu
```

## Result

.config - U-Boot 2016.01 Configuration

U-Boot 2016.01 Configuration

Arrow keys navigate the menu. <Enter> selects submenus ---> (or empty submenus ---). Highlighted letters are hotkeys. Pressing <Y> includes, <N> excludes, <M> modularizes features. Press <Esc><Esc> to exit, <?> for Help, </> for Search. Legend: [\*] built-in [ ]

**Architecture select (ARM architecture) --->**

- ARM architecture --->
- General setup --->
- Boot images --->
- My commands --->
- Android support commands --->
- Command line interface --->
- [ ] Console recording
- Device Tree Control --->
- [\*] Networking support --->

**My commands**

Arrow keys navigate the menu. <Enter> selects submenus ---> (or empty submenus ---). Highlighted letters are hotkeys. Pressing <Y> includes, <N> excludes, <M> modularizes features. Press <Esc><Esc> to exit, <?> for Help, </> for Search. Legend: [\*] built-in [ ]

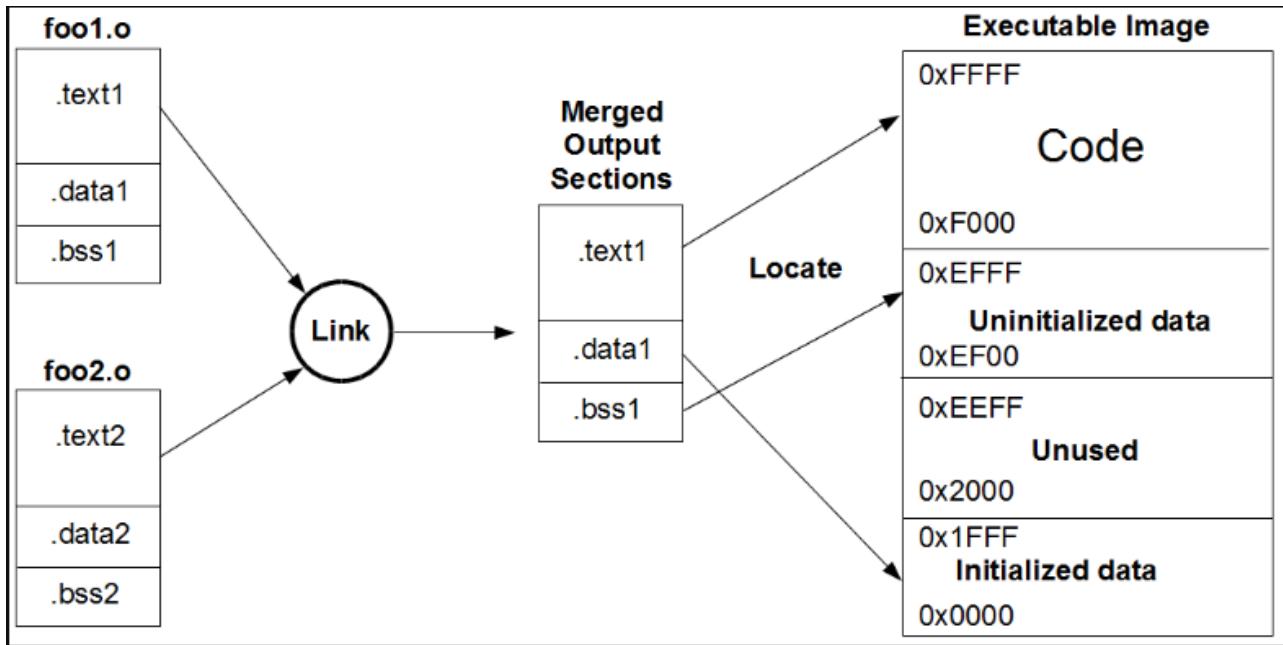
- [\*] Enable 'hello' command

<Select> < Exit > < Help > < Save > < Load >

Rebuild U-Boot and flash it onto the board to check if the command has been successfully added

```
bitminer# hello
Yang Hello_world!
```

## LD script



### What is Linker?

The linker combines multiple object files (produced by the compiler) into a single executable.

### What is an LD Script?

An LD script provides the linker with instructions on how to assemble compiled source files (.o files) into a final executable.

- It specifies the memory layout of the output file, including where sections(chunks of code or data) should be placed in memory, the entry point of the executable, and how to resolve external references.
- The `.lds` file extension denotes a linker script file used by GNU's LD linker

### Why Use an LD Script?

- LD scripts are particularly useful in embedded OS, where precise control over the memory layout of the compiled code is needed.
  - For example, there may be a need to place code or data at specific addresses to match the hardware requirements or bootloading procedures.

## What is Relocation?

Relocation is a process performed by the linker to adjust the addresses of symbolic references(such as variable names or function names) within the code and data sections so that they point to specific memory locations.

## Why is Relocation Needed?

- This process is necessary because, during compilation, the exact addresses of functions, variables, or other data might not be known.
- The compiler generates object code with placeholders(such as variable names or function names) for addresses that cannot be determined until later. The process of relocation involves updating these placeholders with actual addresses.

### example.lds

```
/* Example linker script */
MEMORY
{
    ROM (rx) : ORIGIN = 0x08000000, LENGTH = 256K /* Memory region with read and
execute permissions (rx) for code*/
    RAM (rwx) : ORIGIN = 0x20000000, LENGTH = 64K /* Memory region with read, write,
and execute permissions (rwx) for data */
}

SECTIONS
{
    /* Define the .text section (code) */
    .text : {
        *(.text) /* Include all .text sections from input files */
        *(.rodata) /* Include read-only data */
        . = ALIGN(4); /* Align to 4-byte boundary */
    } > ROM

    /* Define the .data section (initialized data) */
    .data : {
        *(.data) /* Include all .data sections from input files */
        . = ALIGN(4);
    } > RAM AT > ROM /* Place in RAM but initialize from ROM */

    /* Define the .bss section (uninitialized data) */
    .bss : {
        *(.bss)
        *(COMMON)
        . = ALIGN(4);
    }
}
```

```

} > RAM

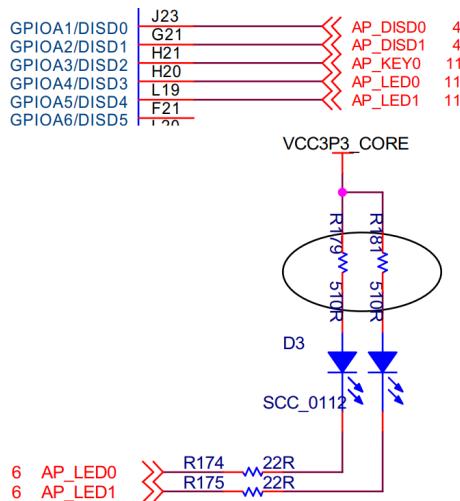
/* Define a symbol at a specific address */
_special_address = 0x20001000;
}

```

### 3. GPIO

Check which GPIO the LED is connected to

GPIOA4 <> AP\_LED0



Device tree

```

gpio_a:gpio@c001a000 {
    compatible = "nexell,nexell-gpio";
    reg = <0xc001a000 0x00000010>;
    altr,gpio-bank-width = <32>;
    gpio-bank-name = "gpio_a";
    gpio-controller;
    #gpio-cells = <2>;
};

```

Data sheet

**Check which memory address the GPIO is connected to**

GPIO\_A starts from 0xc001a000

**16.5.1.1 GPIOxOUT**

- Base Address: C001\_A000h (GPIOA)
- Base Address: C001\_B000h (GPIOB)
- Base Address: C001\_C000h (GPIOC)
- Base Address: C001\_D000h (GPIOD)
- Base Address: C001\_E000h (GPIOE)
- Address = Base Address + A000h, B000h, C000h, D000h, E000h, Reset Value = 0x0000\_0000

Name	Bit	Type	Description	Reset Value
GPIOXOUT	[31:0]	RW	GPIOx[31:0]: Specifies the output value in GPIOx output mode. This bit should be set as "1" (Input mode) or "0" (Output mode) to use the Open drain pins in Input/Output mode. 0 = Low Level 1 = High Level	32'h0

**GPIO control****Command control****Method 1**

Check if defconfig has this line

```
| CONFIG_CMD_GPIO=y
```

Toggle the GPIO the LED is connected to

```
gpio toggle 4
```

**Method 2**

Turn off the LEDs by writing to the memory location

```
md 0xc001a000 # Print
mw 0xc001a000 30 # Write 0x30 = 0b00110000
md 0xc001a000 # Print
```

Before

```
c001a000: 00000000 01000030 00000000 00000000
```



After

```
c001a000: 00000030 01000030 00000000 00000000
```



### Direct control in the u-boot source

Find what function prints this to find the initial booting code

```
Hit any key to stop autoboot:
```

```

SEARCH hit any key
Replace AB ...
Files to include ./common
Files to exclude
2 results in 2 files - Open in editor
C autoboot.c common
print("Hit any key to stop autoboot: %d", bootdelay);
C cmd_bootmenu.c common
printf(" Hit any key to stop autoboot: %d ", menu->delay);

common > C autoboot.c abortboot_normal(int)
201     * To print the bootdelay value upon bootup.
202
203     printf(CONFIG_AUTOBOOT_PROMPT, bootdelay);
204 # endif
205
206     abort = password_abort(etime);
207     if (!abort)
208         | debug_bootkeys("key timeout\n");
209
210 #ifdef CONFIG_SILENT_CONSOLE
211     if (abort)
212         gd->flags &= ~GD_FLG_SILENT;
213 #endif
214
215     return abort;
216 }
217
218 # else /* !defined(CONFIG_AUTOBOOT_KEYED) */
219
220 #ifdef CONFIG_MENUKEY
221     static int menukey;
222 #endif
223
224     static int abortboot_normal(int bootdelay)
225 {
226     int abort = 0;
227     unsigned long ts;
228
229 #ifdef CONFIG_MENUPROMPT
230     printf(CONFIG_MENUPROMPT);
231 #else
232 #ifndef QUICKBOOT
233     if (bootdelay >= 0)
234         printf(" Hit any key to stop autoboot: %d ", bootdelay);
235 // printf("yang(autoboot.c)\n");
236
237     #endif
238 }
239
240 #endif

```

Trace back the found function

```

SEARCH abortboot_normal()
Replace AB ...
Files to include ./common
Files to exclude
2 results in 1 file - Open in editor
C autoboot.c common
static int abortboot_normal(int bootdelay)
    return abortboot_normal(bootdelay);

```

Add the feature

Run this function below `autoboot_command()` in `common/main.c`

```

void blink_LED() {
    int i=0, gpio_value=1;
    for(i=0; i<10; i++){
        printf("hello \n");
        // # Use the device driver functions of the GPIO
        // gpio_request(4, "cmd_gpio");
        // gpio_request(5, "cmd_gpio");
        // gpio_direction_output(4, gpio_value);
        // gpio_direction_output(5, gpio_value);
        // gpio_value=!gpio_value;
}

```

```
// # Bit control
*(char*)0xc001a000 ^= 1 << 4;
*(char*)0xc001a000 ^= 1 << 5;
mdelay(100);
}
```

## 2. KernelBoot

### Difference between kernel images

#### Image

- **Image** refers to the uncompressed, raw binary image of the Linux kernel. This is the most basic form of the kernel image, directly produced by the kernel compilation process. However, this raw Image format is not directly used in most embedded systems due to its size and lack of metadata for bootloaders.

#### zImage

- **zImage** is a compressed version of the Linux kernel(ARM32), built directly from the kernel source. The kernel is compressed to reduce its size, which is crucial for embedded systems with limited storage space. The zImage includes a small decompression stub at the beginning of the image, which is executed by the bootloader and unpacks the kernel into memory before it starts.

#### ulImage

- **ulImage** is a format specific to the U-Boot bootloader, widely used in embedded systems. It wraps around the kernel image (which can be an Image, zImage, or even an initramfs image) with a header that includes metadata like the image's size, load address, entry point, and compression type. This metadata is used for the bootloader to correctly load and execute the kernel image. The ulimage format is designed to be versatile, supporting different types of payloads and compression, making it suitable for a wide range of embedded devices.

## How to compile the linux kernel to make Image and zImage

### Install dependencies

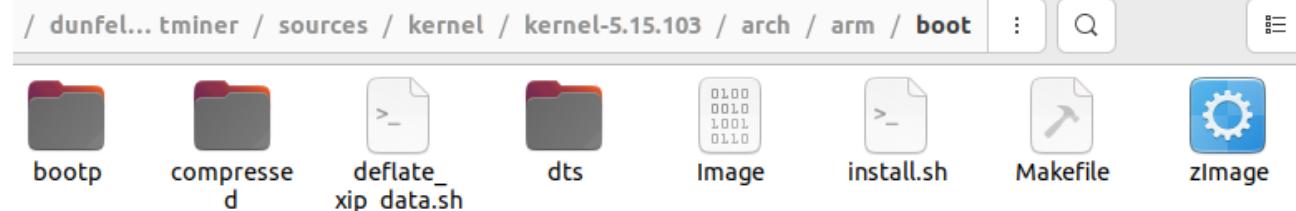
```
sudo apt-get install make build-essential libncurses-dev bison flex libssl-dev
libelf-dev
```

### Build (for ARM32)

```
sudo apt-get install gcc-arm-linux-gnueabihf # Cross compiler
make menuconfig # Save menuconfig and create .config
export ARCH=arm
export CROSS_COMPILE=arm-linux-gnueabihf-
make
# make ARCH=arm CROSS_COMPILE=aarch64-linux-gnu-
```

### Build (for ARM64)

```
sudo apt-get install gcc-aarch64-linux-gnu # Cross compiler
make menuconfig # Save menuconfig and create .config
export ARCH=arm64
export CROSS_COMPILE=aarch64-linux-gnu-
make
# make ARCH=arm64 CROSS_COMPILE=aarch64-linux-gnu-
```



## How to make ulimage

Copy the compiled kernel image to wrap around the ulimage into uboot-2016.01/tools

```
:/work/dunfell-bitminer/sources/kernel/kernel-5.15.103/arch/arm64/boot$ cp Image ~/work/dunfell-bitminer/sources/boot/u-boot/u-boot-2016.01/tools
```

### Create ulimage

```
:/work/dunfell-bitminer/sources/boot/u-boot/u-boot-2016.01/tools$ ./mkimage -A arm64 -O linux -T kernel -C none -a 0x40080000 -e 0x40080000 -d Image uImage
```

```
./mkimage -A arm64 -O linux -T kernel -C none -a 0x40080000 -e 0x40080000 -d Image uImage
```

## How to boot the created image (ulimage in this example)

Check the boot command with \$ printenv

```
boot_cmd_mmcboot=ext4load mmc 0:1 0x40080000 Image;ext4load mmc 0:1 0x49000000 s5p6818-bitminer-rev01.dtb;run dtb_reserve;bootl 0x40080000 - 0x49000000
```

This shows the kernel image is loaded into 0x40080000 and the DTB into 0x49000000

Load the image into the RAM of the board

```
udown 0x40008000 # Enter this in minicom to prepare to load the image into the RAM of the board
./usb-downloader -t slsiap -f ~/work/dunfell-bitminer/sources/boot/u-boot/u-boot-2016.01/tools/uImage
udown 0x49000000 # for the DTB
./usb-downloader -t slsiap -f ~/work/dunfell-bitminer/build_bitminer_bitminer/tmp/deploy/images/s5p6818/s5p6818-bitminer-rev01.dtb
```

```
bitminer# udown 0x40008000
Download Address 40008000 Size 20429376(hex : 137ba40)
Download complete
```

```
USB Download Tool

=> Header Mode
=====
Nexell USB Downloader Version 1.4.2-artik, Mar  3 2017, 15:36:32
Send Image width NSHI Header
processor type  : slsiap
nsih file      : NULL
bin file       : /home/youngsik/work/dunfell-bitminer/sources/boot/u-boot/u-bo
ot-2016.01/tools/uImage
secondboot file : NULL
download addr   : default
start addr     : default
=> Downloading 20429888 bytes
=> Download Success!!!
```

Run the ulimage in the target board

```
bootm 0x40008000 - 0x49000000
```

```

bitminer# bootm 0x40008000 - 0x49000000
## Booting kernel from Legacy Image at 40008000 ...
  Image Name:
  Image Type: AArch64 Linux Kernel Image (uncompressed)
  Data Size: 20429312 Bytes = 19.5 MiB
  Load Address: 40080000
  Entry Point: 40080000
  Verifying Checksum ... OK
## Flattened Device Tree blob at 49000000
  Booting using the fdt blob at 0x49000000
  Loading Kernel Image ... Image too large: increase CONFIG_SYS_BOOTM_LEN
Must RESET board to recover
resetting ...
b*
U-BootDRAM: 219 MiB
POFFHIS: NONE
PONHIS : PWRONPON
CHGS : CHG OFF
DCDC1 : 1250mV, En, dclim:1303023664, dclimsden:0
DCDC2 : 1200mV, En, dclim:1303023664, dclimsden:0
DCDC3 : 3300mV, En, dclim:1303023664, dclimsden:0
DCDC4 : 1500mV, En, dclim:1303023664, dclimsden:0
DCDC5 : 1500mV, En, dclim:1303023664, dclimsden:0
MMC: NEXELL DWMMC: 0
In: serial
Out: serial
Err: serial
Net:
Warning: ethernet@c0060000 (eth0) using random MAC address - e6:44:71:53:be:f9
eth0: ethernet@c0060000
Hit any key to stop autoboot: 0
11926784 bytes read in 561 ms (20.3 MiB/s)
41888 bytes read in 8 ms (5 MiB/s)
## Error: "dtb_reserve" not defined
## Flattened Device Tree blob at 49000000
  Booting using the fdt blob at 0x49000000
  Using Device Tree in place at 0000000049000000, end 000000004900d39f

Starting kernel ...

```

## booti, bootz

- booti: The default boot command of this target board is booti to boot an uncompressed Image. (e.g. booti 0x40008000 - 0x49000000)
- bootz: bootz is disabled by default and should be enabled if it needs to be used. (e.g. bootz 0x40008000 - 0x49000000)

Add this in common/kconfig to enable bootz

```

config CMD_BOOTZ
    bool "bootz"
    default y
    help
        Boot a zImage from the memory.
)

```

## How to upload data into the eMMC using fastboot

### File system



boot.img

- **Kernel Image ( Image )**: This is the compiled Linux kernel. It's the core of the operating system that manages the hardware and allows other programs to run.
- **Device Tree Blob ( \*.dtb )**: This file describes the hardware in the system so that the kernel knows what devices are present and how they are configured.
- **Ramdisk ( initrd )**: This is a temporary root file system used during the boot process before the real root filesystem is mounted.

These components are combined into a flashable boot.img to get flashed onto the file system of the eMMC

### How to check if boot.img contains the kernel Image

```

def find_pattern(source, pattern):
    position = source.find(pattern)
    if position != -1:
        print(f"Found the pattern at position: {position}")
    else:
        print("The pattern not found in the file.")

with open('boot.img', 'rb') as f:
    boot_img = bytearray(f.read())
with open('Image', 'rb') as f2:
    kernel = bytearray(f2.read())
    find_pattern(boot_img, kernel)
    kernel[1] = 1
    find_pattern(boot_img, kernel)

```

```

Found the pattern at position: 5300372
The pattern not found in the file.

```

## partmap\_emmc.txt

```

1 flash=mmc,0:2ndboot:2nd:0x200,0x10000:bl1-emmcboot.img;
2 flash=mmc,0:fip-loader:boot:0x10200,0x50000:fip-loader-emmc.img;
3 flash=mmc,0:fip-secure:boot:0x60200,0x180000:fip-secure.img;
4 flash=mmc,0:fip-nonsecure:boot:0x1E0200,0x100000:fip-nonsecure.img;
5 flash=mmc,0:env:env:0x2E0200,0x4000:params.bin;
6 flash=mmc,0:boot:ext4:0x400000,0x4000000:boot.img;
7 flash=mmc,0:rootfs:ext4: 0x4400000,0x3b400000:rootfs.img;
8 flash=mmc,0:user:ext4: 0x40000000,0:userdata.img;|

```

## Upload

### Host PC

```

sudo fastboot flash partmap boot-binary/partmap_emmc.txt # Map partitions in the eMMC
# Upload
sudo fastboot flash 2ndboot boot-binary/bl1-emmcboot.img
sudo fastboot flash fip-loader boot-binary/fip-loader-emmc.img
sudo fastboot flash fip-secure boot-binary/fip-secure.img
sudo fastboot flash fip-nonsecure boot-binary/fip-nonsecure.img
sudo fastboot flash env boot-binary/params.bin
sudo fastboot flash boot boot.img
sudo fastboot flash rootfs rootfs.img
sudo fastboot flash user userdata.img

```

### Target board (in u-boot)

```
fast 0
```

## 3. MMU

### Enable cache

common/cmd\_cache.c for the cache commands

cmd\_cache.c

## Compile cmd\_cache.c

```
common/Makefile
obj-$(CONFIG_CMD_CACHE) += cmd_cache.o
```

## Enable the cache commands

```
menu my_commands
config CMD_CACHE
    bool "cache"
    default y
    help
        cache
```

## Enable dcache(disabled by default)

This doesn't work

```
config SYS_DCACHE_OFF
    bool "Do not use Data Cache"
    default n
```

In this file

```
u-boot-2016.01 > include > configs > s5p6818_bitminer.h
```

Change this

```
/* board_init_f, CONFIG_SYS_ICACHE_OFF */
#define CONFIG_SYS_DCACHE_OFF
```

to this

```
/* board_init_f, CONFIG_SYS_ICACHE_OFF */
// #define CONFIG_SYS_DCACHE_OFF
```

Check icache and dcache are enabled

```
bitminer# icache
Instruction Cache is ON
bitminer# dcache
Data (writethrough) Cache is ON
```

## Check the speed when the cache is on and off

### icache(Instruction cache)

1. icache on
2. mw.l 0x7A000000 0x00FF0000 614400 (This is fast)
3. icache off
4. mw.l 0x7A000000 0x00FF0000 614400 (This is slow)

### dcache(Data cache)

```
static void do_test_cache(void)
{
    ulong start_time, end_time;
    printf("Testing cache speed...\n");
    start_time = get_timer(0); // Start timer
    unsigned long long i=1, result=0;
    for(i=1; i<=87654321; i++)
        result += i % 2;
    end_time = get_timer(start_time); // End timer
    printf("Time taken: %lu ms\n", end_time);
    printf("Result: %d\n", result);
}
```

```
bitminer# dcache on
bitminer# test_cache
Testing cache speed...
Time taken: 328 ms
Result: 43827161
bitminer# dcache off
bitminer# test_cache
Testing cache speed...
Time taken: 1095 ms
Result: 43827161
```

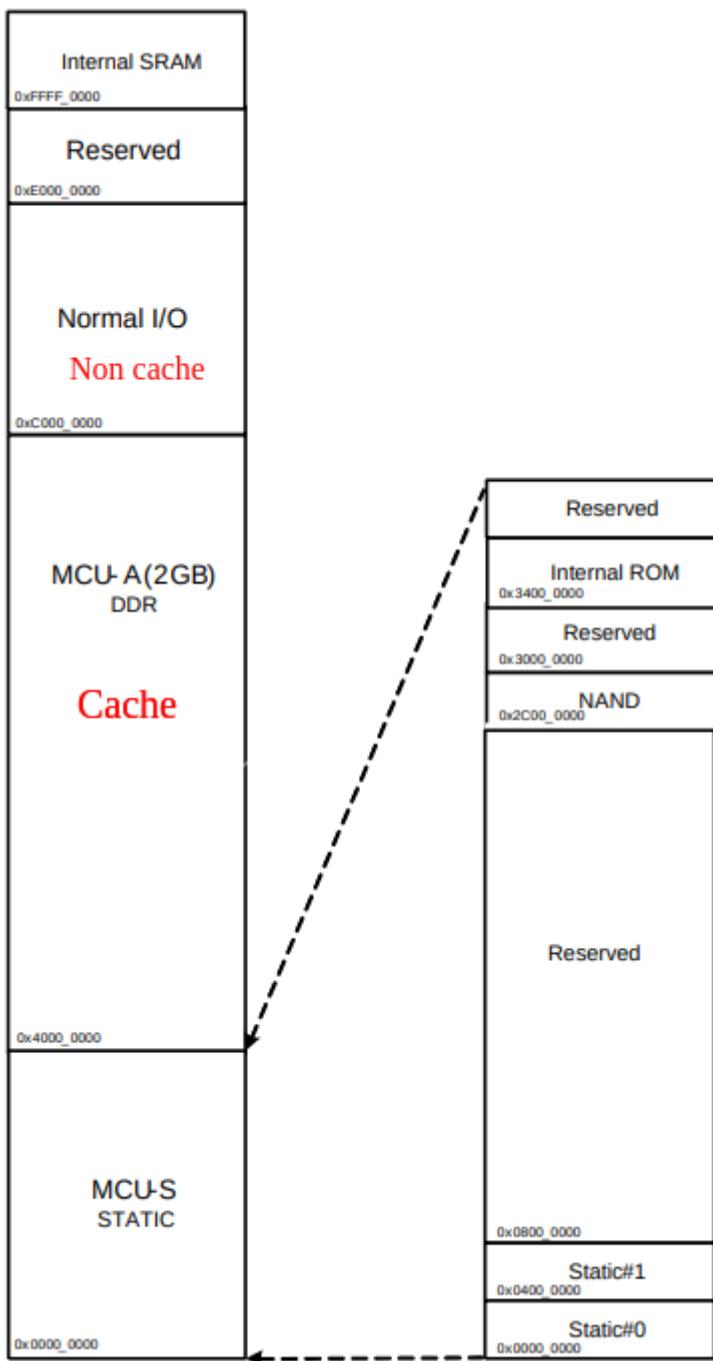
## Cache in MMU page table

arch/arm/lib/cache-cp15.c

```
_weak void dram_bank_mmu_setup(int bank)
{
    debug("%s: bank: %d\n", __func__, bank);
    for (i = bd->bi_dram[bank].start >> 20;
         i < (bd->bi_dram[bank].start >> 20) + (bd->bi_dram[bank].size >> 20);
         i++) {
#if defined(CONFIG_SYS_ARM_CACHE_WRITETHROUGH)
        set_section_dcache(i, DCACHE_WRITETHROUGH);
#elif defined(CONFIG_SYS_ARM_CACHE_WRITEALLOC)
        set_section_dcache(i, DCACHE_WRITEALLOC);
#else
        set_section_dcache(i, DCACHE_WRITEBACK);
#endif
    }
}
```

This code modifies cache attributes in MMU's page table entries

## Cache in memory map



- SRAM is a cache itself
- DRAM is cacheable for faster speed
- Memory-mapped I/O should be non-cacheable for data consistency (Bits in memory-mapped I/O should not be stored in cache because cache is not mapped to the I/O targets)

## 4. Device Driver assignment

### INDEX

---

### 0. UPDATE HISTORY

EVENT	Date	Description	Author
create	2024.2.2 0	create	@NXKR_YoungSikYan g
modify	2024.2.2 1	modified the contents according to the template	@NXKR_YoungSikYan g
modify	2024.2.2 1	<ul style="list-style-type: none"> <li>corrected wrong indentation</li> <li>added source code</li> <li>added gpio-uclass.c reference</li> </ul>	@NXKR_YoungSikYan g
modify	2024.2.2 2	<ul style="list-style-type: none"> <li>modified implemented features</li> <li>deleted memory location</li> <li>modified test scenario</li> </ul>	@NXKR_YoungSikYan g
modify	2024.2.2 3	<ul style="list-style-type: none"> <li>Modified command manual so that it can be clearer</li> <li>Corrected a typo in flow diagram</li> <li>Added more explanation to each section</li> </ul>	@NXKR_YoungSikYan g

### 1. Introduction

- Final assignment of u-boot training.
- Board: BTC08(s5p6818)

Utilizing the GPIO device driver implemented in gpio-uclass.c, I have developed features as specified in the assignment requirements and tested them using u-boot commands.

## 2. Implemented features

The features are related to controlling or querying GPIOs.

### 2.1 Feature list

The features can be classified into 3 categories

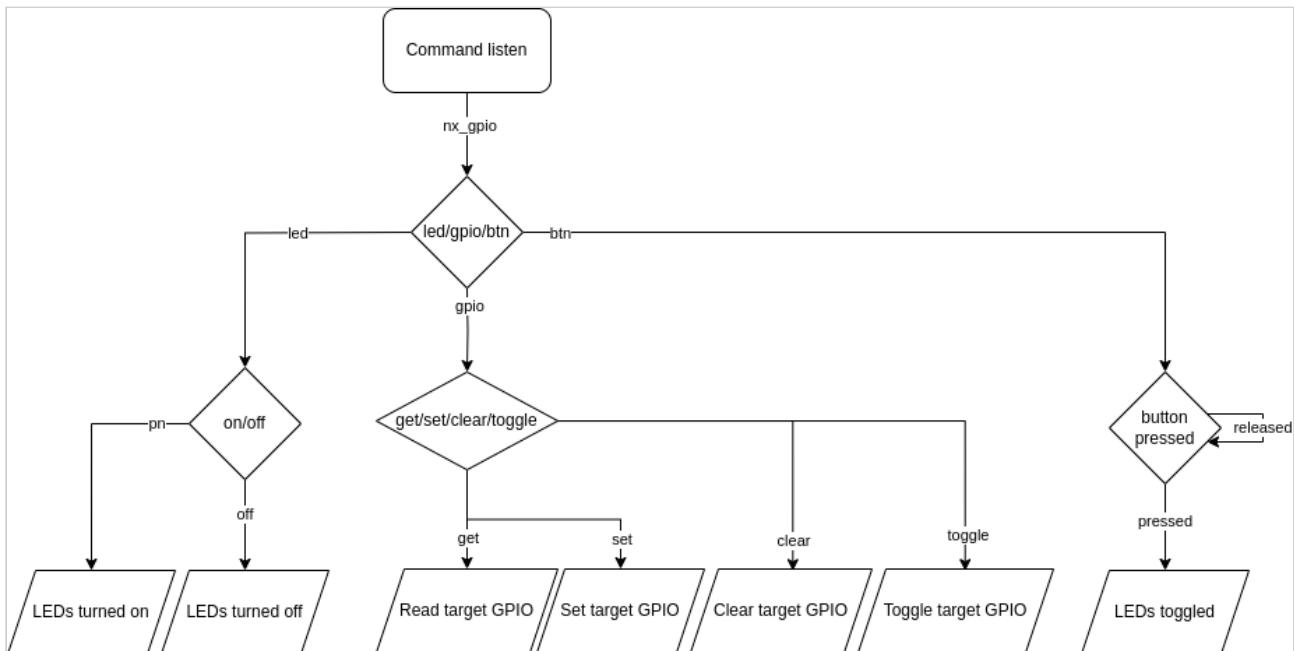
- GPIO input/output (GPIOA, GPIOB, GPIOC, GPIOD, GPIOE)
- LED control (AP\_LED0, AP\_LED1)
- Button detection (AP\_KEY0)

### 2.2 Command manual

Command	Action
nx_gpio led <on off>	Turn the LEDs on or off
nx_gpio <set clear toggle get> <num>	set clear toggle get <num> GPIO
nx_gpio btn	Wait for button presss and toggle the LEDs

### 2.3 Flow diagram

This diagram visualizes the behavior of each command.



### 3. Target GPIO pins

This picture is from Page 6 of tsb1101\_control\_v100\_rel\_20181126

GPIOA3/DISD2	H20	AP_KEY0	11
GPIOA4/DISD3	L19	AP_LED0	11
GPIOA5/DISD4	F21	AP_LED1	11



## 4. Source

### 4.1 Source code

#### common/cmd\_nx\_gpio.c

```
#include <common.h>
#include <command.h>
#include <stdarg.h>
#include <errno.h>
#include <dm.h>
#include <asm/gpio.h>

#define BTN 3
#define LED0    4
#define LED1    5
#define ON     0
#define OFF   1

static int gpios_toggle(int gpio, ...)
{
    va_list args;
    va_start(args, gpio);
    int ret;
    while (gpio != -1) {
        if (gpio_request(gpio, "cmd_nx_gpio")) {
            printf("Failed to request gpio %i\n", gpio);
            return -1;
        }
        ret = gpio_get_value(gpio);
        if (ret < 0) {
            printf("Failed to get gpio %i value\n", gpio);
            gpio_free(gpio);
            return ret;
        }
        ret = gpio_set_value(gpio, !ret);
        if (ret < 0) {
            printf("Failed to toggle gpio %i value\n", gpio);
            gpio_free(gpio);
            return ret;
        }
        printf("gpio %i value is %d\n", gpio, !ret);
        gpio_free(gpio);
        gpio = va_arg(args, int);
    }
    va_end(args);
}
```

```

    return ret;
}

static int gpios_set_value(int gpio_value, int gpio, ...)
{
    va_list args;
    va_start(args, gpio);
    int ret;
    while (gpio != -1) {
        if (gpio_request(gpio, "cmd_nx_gpio")) {
            printf("Failed to request gpio %i\n", gpio);
            return -1;
        }
        ret = gpio_set_value(gpio, gpio_value);
        if (ret < 0) {
            printf("Failed to set gpio %i value\n", gpio);
            gpio_free(gpio);
            return ret;
        }
        printf("gpio %i value is %d\n", gpio, gpio_value);
        gpio_free(gpio);
        gpio = va_arg(args, int);
    }
    va_end(args);
    return ret;
}

int do_nx_gpio(cmd_tbl_t *cmdtp, int flag, int argc, char * const argv[])
{
    int gpio_value, gpio, ret;
    /* Not enough args */
    if (argc < 2) {
        printf("Usage: %s\n", cmdtp->usage);
        return CMD_RET_FAILURE;
    }

    if (strcmp(argv[1], "led") == 0) {
        if (argc < 3) {
            printf("Usage: %s\n", cmdtp->usage);
            return CMD_RET_FAILURE;
        }
        if (strcmp(argv[2], "on") == 0) {
            if (gpios_set_value(ON, LED0, LED1, -1))
                return CMD_RET_FAILURE;
        } else if (strcmp(argv[2], "off") == 0) {
            if (gpios_set_value(OFF, LED0, LED1, -1))
                return CMD_RET_FAILURE;
        } else {
            printf("Usage: %s\n", cmdtp->usage);
            return CMD_RET_FAILURE;
        }
    } else if (strcmp(argv[1], "gpio") == 0) {

```

```

if (argc < 4) {
    printf("Usage: %s\n", cmdtp->usage);
    return CMD_RET_FAILURE;
}
ret = gpio_lookup_name(argv[3], NULL, NULL, &gpio);
if (gpio < 0) {
    printf("Usage: %s\n", cmdtp->usage);
    return CMD_RET_FAILURE;
}
if (strcmp(argv[2], "get") == 0) {
    if (gpio_request(gpio, "cmd_nx_gpio")) {
        printf("Failed to request gpio %i\n", gpio);
        return CMD_RET_FAILURE;
    }
    gpio_value = gpio_get_value(gpio);
    if (gpio_value < 0) {
        printf("Failed to get gpio %i value\n", gpio);
        gpio_free(gpio);
        return CMD_RET_FAILURE;
    }
    printf("gpio %i value is %d\n", gpio, gpio_value);
    gpio_free(gpio);
} else if (strcmp(argv[2], "set") == 0) {
    if (gpios_set_value(1, gpio, -1))
        return CMD_RET_FAILURE;
} else if (strcmp(argv[2], "clear") == 0) {
    if (gpios_set_value(0, gpio, -1))
        return CMD_RET_FAILURE;
} else if (strcmp(argv[2], "toggle") == 0) {
    if (gpios_toggle(gpio, -1))
        return CMD_RET_FAILURE;
} else {
    printf("Usage: %s\n", cmdtp->usage);
    return CMD_RET_FAILURE;
}
} else if (strcmp(argv[1], "btn") == 0) {
    printf("Waiting for button press...\n");
    if (gpio_request(BTN, "cmd_nx_gpio")) {
        printf("Failed to request gpio %i\n", gpio);
        return CMD_RET_FAILURE;
    }
/* Polling */
while (1) {
    gpio_value = gpio_get_value(BTN);
    if (gpio_value < 0) {
        printf("Failed to get gpio %i value\n", 3);
        gpio_free(BTN);
        return CMD_RET_FAILURE;
    }
    if (gpio_value == 0) {
        gpio_free(BTN);
        break;
    }
}

```

```

        }
    }
    /* Toggle LEDs */
    if (gpios_toggle(LED0, LED1, -1))
        return CMD_RET_FAILURE;
} else {
    printf("Usage: %s\n", cmdtp->usage);
    return CMD_RET_FAILURE;
}
return CMD_RET_SUCCESS;
}

U_BOOT_CMD(nx_gpio, 4, 0, do_nx_gpio,
           "Query and control gpio pins",
           "\n- led on|off: Turn the LEDs on|off\n"
           "- gpio set|clear|toggle|get <num>: Set|Clear|Toggle|Get <num> gpio\n"
           "- btn: Wait for button press and toggle 2 LEDs\n"
);

```

## 4.2 Add the source code

Configure so that the new source code can be compiled and included in u-boot through the build process.

**common/Makefile**

This adds the source to the compile list.

```
# My code
obj-$(CONFIG_CMD_NX_GPIO) += cmd_nx_gpio.o
obj-$(CONFIG_TEST_CMD_NX_GPIO) += test/cmd_nx_gpio.o
```

**common/kconfig**

This includes the source in u-boot.

```
config CMD_NX_GPIO
    bool "nx_gpio"
    default y
    help
        Turn on custom GPIO commands
config TEST_CMD_NX_GPIO
    bool "Test nx_gpio commands"
    default y
    help
        Test for custom GPIO commands
```

And then build

## 4.3 Existing APIs used in the source code

The source code above takes advantage of this API source

<https://github.com/NexellCorp/u-boot-2016.01/blob/artik/drivers/gpio/gpio-uclass.c>

## 5. Test

Each command is tested by visually checking the color of the LEDs and verifying the input of the target GPIO through the GPIO driver.

### 5.1 Test scenario

Case	Command	Expectation	Goal
get GPIOA3	nx_gpio gpio get 3	GPIOA3 is high with button released → button press -> GPIOA3 is low	GPIO input works
led off	nx_gpio led off	LEDs are on → LEDs are turned off	LED on/off works
led on	nx_gpio led on	LEDs are off → LEDs are turned on	LED on/off works
led abc	nx_gpio led abc	Command does not work (invalid command)	Invalid command does not work
set GPIOA4,5	nx_gpio gpio set 4 nx_gpio gpio set 5	GPIOA4,5 are 0 → GPIOA4,5 are set to 1(LEDs are turned off)	GPIO output works
clear GPIOA4,5	nx_gpio gpio clear 4 nx_gpio gpio clear 5	GPIOA4,5 are 1 → GPIOA4,5 are cleared to 0(LEDs are turned on)	GPIO output works
toggle GPIOA4,5	nx_gpio gpio toggle 4 nx_gpio gpio toggle 5	GPIOA4,5 are 0 → GPIOA4,5 are toggled to 1(LEDs are turned off)	GPIO output works

toggle GPIOA4,5 on button press	nx_gpio btn	GPIOA4,5 are 1 → button press → GPIOA4,5 are toggled to 0(LEDs are turned on)	GPIO output works
---------------------------------	-------------	---	-------------------

## 5.2 Test code

### test/cmd\_nx\_gpio.c

```
#include <common.h>
#include <command.h>
#include <errno.h>
#include <dm.h>
#include <asm/gpio.h>

static void gpio_get(unsigned int *total_tests, unsigned int *passed_tests)
{
    printf("Test: gpio get\n");
    *total_tests += 1;
    /* Given: button connected to GPIO 3 is released */
    /* When: check the value of GPIO 3 */
    gpio_request(3, "cmd_nx_gpio");
    char gpio_value = gpio_get_value(3);
    /* Then: GPIO 3 is high */
    if (gpio_value == 1)
        printf("GPIO 3 is high.(released)\n");
    else {
        printf("Test failed: GPIO 3 is not high.\n");
        return;
    }
    printf("Waiting for button press...\n");
    /* Given: button connected to GPIO 3 is released */
    /* When: button is pressed */
    while (1) {
        gpio_value = gpio_get_value(3);
        if (gpio_value == 0)
            break;
    }
    /* Then: GPIO 3 is low */
    if (gpio_get_value(3) == 0) {
        printf("Test passed: GPIO 3 is low.(pressed)\n");
        ++passed_tests;
    } else
        printf("Test failed: GPIO 3 is not low.\n");
    gpio_free(3);
}
```

```

static void led_off(unsigned int *total_tests, unsigned int *passed_tests)
{
    printf("Test: led off\n");
    *total_tests += 1;
    /* Given: LEDs are on */
    /* When: turn off the LEDs */
    run_command("nx_gpio led off", 0);
    /* Then: GPIO 4 and 5 are high */
    gpio_request(4, "cmd_nx_gpio");
    gpio_request(5, "cmd_nx_gpio");
    if (gpio_get_value(4) == 1 && gpio_get_value(5) == 1) {
        printf("Test passed: LEDs off.\n");
        ++passed_tests;
    } else
        printf("Test failed: LEDs are not off.\n");
    gpio_free(4);
    gpio_free(5);
}

static void led_on(unsigned int *total_tests, unsigned int *passed_tests)
{
    printf("Test: led on\n");
    *total_tests += 1;
    /* Given: LEDs are off */
    /* When: turn on the LEDs */
    run_command("nx_gpio led on", 0);
    /* Then: GPIO 4 and 5 are low */
    gpio_request(4, "cmd_nx_gpio");
    gpio_request(5, "cmd_nx_gpio");
    if (gpio_get_value(4) == 0 && gpio_get_value(5) == 0) {
        printf("Test passed: LEDs on.\n");
        ++passed_tests;
    } else
        printf("Test failed: LEDs are not on.\n");
    gpio_free(4);
    gpio_free(5);
}

static void led_abc(unsigned int *total_tests, unsigned int *passed_tests)
{
    int ret;
    printf("Test: led abc\n");
    *total_tests += 1;
    /* Given: LEDs are off */
    /* When: turn on the LEDs */
    ret = run_command("nx_gpio led abc", 0);
    /* Then: Invalid command */
    if (ret == 1) {
        printf("Test passed: Invalid command.\n");
        ++passed_tests;
    } else if (ret == 0)
        printf("Test failed: Command not invalid.\n");
}

```

```

}

static void gpio_set(unsigned int *total_tests, unsigned int *passed_tests)
{
    printf("Test: gpio set\n");
    *total_tests += 1;
    /* Given: LEDs are on */
    /* When: setting GPIO 4 and 5 */
    run_command("nx_gpio gpio set 4", 0);
    run_command("nx_gpio gpio set 5", 0);
    /* Then: LEDs are off */
    gpio_request(4, "cmd_nx_gpio");
    gpio_request(5, "cmd_nx_gpio");
    if (gpio_get_value(4) == 1 && gpio_get_value(5) == 1) {
        printf("Test passed: GPIO 4 and 5 set.\n");
        +++passed_tests;
    } else
        printf("Test failed: GPIO 4 and 5 are not set.\n");
    gpio_free(4);
    gpio_free(5);
}

static void gpio_clear(unsigned int *total_tests, unsigned int *passed_tests)
{
    printf("Test: gpio clear\n");
    *total_tests += 1;
    /* Given: LEDs are off */
    /* When: clear GPIO 4 and 5 */
    run_command("nx_gpio gpio clear 4", 0);
    run_command("nx_gpio gpio clear 5", 0);
    /* Then: LEDs are on */
    gpio_request(4, "cmd_nx_gpio");
    gpio_request(5, "cmd_nx_gpio");
    if (gpio_get_value(4) == 0 && gpio_get_value(5) == 0) {
        printf("Test passed: GPIO 4 and 5 cleared.\n");
        +++passed_tests;
    } else
        printf("Test failed: GPIO 4 and 5 are not cleared.\n");
    gpio_free(4);
    gpio_free(5);
}

static void gpio_toggle(unsigned int *total_tests, unsigned int *passed_tests)
{
    printf("Test: gpio toggle\n");
    *total_tests += 1;
    /* Given: initial state of GPIO 4 and 5 */
    gpio_request(4, "cmd_nx_gpio");
    gpio_request(5, "cmd_nx_gpio");
    char gpio4_value = gpio_get_value(4);
    char gpio5_value = gpio_get_value(5);
    gpio_free(4);
}

```

```

gpio_free(5);
/* When: toggle GPIO 4 and 5 */
run_command("nx_gpio gpio toggle 4", 0);
run_command("nx_gpio gpio toggle 5", 0);
gpio_free(4);
gpio_free(5);
/* Then: GPIO 4 and 5 are toggled */
gpio_request(4, "cmd_nx_gpio");
gpio_request(5, "cmd_nx_gpio");
if (gpio4_value != gpio_get_value(4)
&& gpio5_value != gpio_get_value(5)) {
    printf("Test passed: GPIO 4 and 5 toggled.\n");
    ++passed_tests;
} else
    printf("Test failed: GPIO 4 and 5 are not toggled.\n");
gpio_free(4);
gpio_free(5);
}

static void btn(unsigned int *total_tests, unsigned int *passed_tests)
{
    printf("Test: btn\n");
    *total_tests += 1;
    /* Given: initial state of GPIO 4 and 5 /
     the button connected to GPIO 3 is released */
    gpio_request(4, "cmd_nx_gpio");
    gpio_request(5, "cmd_nx_gpio");
    char gpio4_value = gpio_get_value(4);
    char gpio5_value = gpio_get_value(5);
    gpio_free(4);
    gpio_free(5);
    /* When: wait for button to be pressed */
    run_command("nx_gpio btn", 0);
    /* Then: LEDs are toggled */
    gpio_request(4, "cmd_nx_gpio");
    gpio_request(5, "cmd_nx_gpio");
    if (gpio4_value != gpio_get_value(4) &&
    gpio5_value != gpio_get_value(5)) {
        printf("Button pressed and LEDs toggled.\n");
        ++passed_tests;
    } else
        printf("Button press not recognized or LEDs not toggled.\n");
    gpio_free(4);
    gpio_free(5);
}

static void run_test(void (*test)(unsigned int *, unsigned int *),
unsigned int *total_tests, unsigned int *passed_tests)
{
    test(total_tests, passed_tests);
    printf("Press any key for the next test...-----\n");
    while (!tstc())
}

```

```

        ;
getc();
}

int do_test_nx_gpio(void)
{
    unsigned int total_tests = 0;
    unsigned int passed_tests = 0;
    run_test(gpio_get, &total_tests, &passed_tests);
    if (passed_tests == 0) /* Other tests depend on this one */
        return;
    run_test(led_off, &total_tests, &passed_tests);
    run_test(led_on, &total_tests, &passed_tests);
    run_test(led_abc, &total_tests, &passed_tests);
    run_test(gpio_set, &total_tests, &passed_tests);
    run_test(gpio_clear, &total_tests, &passed_tests);
    run_test(gpio_toggle, &total_tests, &passed_tests);
    btn(&total_tests, &passed_tests);
    printf("-----\n"
"Test completed. %u out of %u tests passed.\n",
passed_tests, total_tests);
    return CMD_RET_SUCCESS;
}

U_BOOT_CMD(test_nx_gpio, 1, 0, do_test_nx_gpio,
"Test cmd_nx_gpio",
"\nRun tests for cmd_nx_gpio\n"
", which include gpio get, led on|off, gpio set|clear|toggle, and btn\n"
);

```

### 5.3 Run test

```
$ test_nx_gpio
```

### 5.4 Test result

The result table is below.

```
bitminer# test_nx_gpio
Test: gpio get
GPIO 3 is high.(released)
Waiting for button press...
Test passed: GPIO 3 is low.(pressed)
Press any key for the next test...
-----
Test: led off
gpio 4 value is 1
gpio 5 value is 1
Test passed: LEDs off.
Press any key for the next test...
-----
Test: led on
gpio 4 value is 0
gpio 5 value is 0
Test passed: LEDs on.
Press any key for the next test...
-----
Test: led abc
Usage: Query and control gpio pins
Test passed: Invalid command.
Press any key for the next test...
-----
Test: gpio set
gpio 4 value is 1
gpio 5 value is 1
Test passed: GPIO 4 and 5 set.
Press any key for the next test...
```

```

Test: gpio clear
gpio 4 value is 0
gpio 5 value is 0
Test passed: GPIO 4 and 5 cleared.
Press any key for the next test...
-----
Test: gpio toggle
gpio 4 value is 1
gpio 5 value is 1
Test passed: GPIO 4 and 5 toggled.
Press any key for the next test...
-----
Test: btn
Waiting for button press...
gpio 4 value is 1
gpio 5 value is 1
Button pressed and LEDs toggled.
-----
Test completed. 8 out of 8 tests passed.

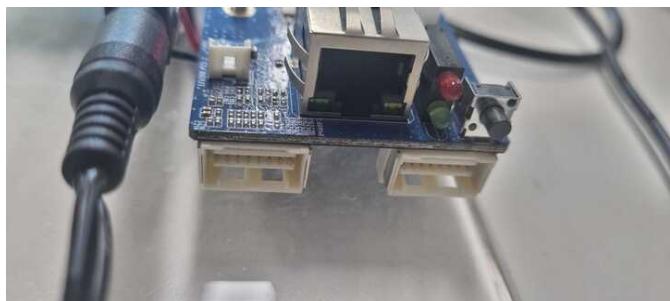
```

**Test result table**

- **Pass:** Test case worked as "expectation" of the test scenario table
- **Fail:** Test case did not work as "expectation" of the test scenario table

Case	Result
get GPIOA3 input	Pass
led off	Pass
led on	Pass
led abc	Pass
set GPIOA4,5	Pass
clear GPIOA4,5	Pass
toggle GPIOA4,5	Pass
toggle GPIOA4,5 on button press	Pass

**LEDs successfully turned on and off before and after each GPIO output command**



## Reference

- tsb1101\_control\_v100\_rel\_20181126
- S5P6818\_Datasheet\_Ver142A
- <https://docs.u-boot.org/en/latest/>
- u-boot-2016.01

# 양영식

## ARM 양영식

### 1. Introduction to the ARM Architecture.

#### Introduction to the ARM Architecture

##### A1.1 About the ARM architecture

###### Key Characteristics

ARM is a Reduced Instruction Set Computer (**RISC**) architecture, which includes:

- Large uniform register file.(These registers are uniform, meaning they can all be used interchangeably for a wide range of instructions. This uniformity simplifies the instruction set and improves the speed of computation.)
- Load/store architecture.(Data-processing operations (like addition, subtraction, etc.) are performed only on data in the registers, not directly on data in memory.)
- Simple addressing modes.(Addressing modes refer to the ways in which instructions specify the location of data. The smaller number of simple addressing modes makes it easier to design the hardware to decode and execute instructions, which can lead to faster processing speeds.)
- Uniform and fixed-length instruction fields, to simplify instruction decode.(to facilitate pipeline)

###### Enhancements

- Control over the Arithmetic Logic Unit (ALU) and shifter.
- Auto-increment and auto-decrement addressing modes.
- Load and Store Multiple instructions.
- Conditional execution of almost all instructions.

These enhancements to a basic RISC architecture allow ARM processors to achieve a good balance of high performance, small code size, low power consumption, and small silicon area.

###### Coprocessor

Coprocessors can be attached to the ARM processor. A coprocessor extends the processing features of a core by extending the instruction set or by providing configuration registers.

### A1.1.1 ARM Registers

- Total of 37 registers
  - 31 general-purpose 32-bit registers, including a program counter. R0 ~ R15 visible at any time.
  - 6 status registers

### A1.1.2 Exceptions

- **Types of Exceptions:**
  - reset
  - attempted execution of an Undefined instruction
  - software interrupt (SWI) instructions, can be used to make a call to an operating system
  - Prefetch Abort, an instruction fetch memory abort
  - Data Abort, a data access memory abort
  - IRQ, Interrupt request
  - FIQ, Fast interrupt request

## A1.2 ARM Instruction Set

### Classification

1. **Branch Instructions**
2. **Data-Processing Instructions**
3. **Status Register Transfer Instructions**
4. **Load and Store Instructions**
5. **Coprocessor Instructions**
6. **Exception-Generating Instructions**

### Features

- Conditional execution based on condition code flags.
- Most data-processing instructions can update the four condition code flags in the CPSR (Negative, Zero, Carry and oVerflow) according to their result.

### A1.2.1 Branch Instructions

Used for control flow alteration by writing the PC(Program Counter)

- **Standard Branch Instruction:** Uses a 24-bit signed word offset for forward and backward branches.

- **Branch and Link (BL):** Preserves return address in the Link Register.
- **Instruction Set Switching:** Allows switching between ARM and Thumb instruction sets.

#### A1.2.2 Data-Processing Instructions

- **Arithmetic/Logic Instructions:** Perform operations on two source operands and write the result.
- **Comparison Instructions:** Perform operations without writing the result, just updating condition flags.
- **SIMD Instructions:** Single Instruction, Multiple Data) Treat operands as parallel 16-bit or 8-bit numbers (ARMv6).
- **Multiply Instructions:** Various classes for different operations.

#### A1.2.3 Status Register Transfer Instructions

Transfer contents between CPSR/SPSR and general-purpose registers.

#### A1.2.4 Load and Store Instructions

Move data between memory and registers.

- **Types:** Regular, Multiple registers, Exclusive.
- **Addressing Modes:** Offset, Pre-indexed, Post-indexed.
- **Support for Unaligned Accesses:** Introduced in ARMv6.

#### A1.2.5 Coprocessor Instructions

- **Types:** Data-processing, Data transfer, Register transfer.

#### A1.2.6 Exception-Generating Instructions

- **Software Interrupt (SWI):** Used for OS-defined services(software interrupt).
- **Software Breakpoint (BKPT):** Causes an abort exception for debugging.

### A1.3 Thumb Instruction Set

A subset of ARM instruction set with 16-bit encoding for instructions. It offers code density improvements. (Code density refers to the amount of instructions that can be packed into a given amount of memory space.)

## 2. Programmers' model

### Programmers' Model

#### Data Types

- **Byte (8 bits)**: Smallest data type.
- **Halfword (16 bits)**: Introduced in ARM version 4.
- **Word (32 bits)**: Main data type for operations.
- **Support for Unaligned Data**: ARMv6 introduced unaligned data support for words and halfwords.
- **Signed and Unsigned Types**: Represent integers using normal binary format or two's complement format.

#### Processor Modes

Table A2-1 ARM processor modes

Processor mode	Mode number	Description
User	usr	0b10000
FIQ	fiq	0b10001
IRQ	irq	0b10010
Supervisor	svc	0b10011
Abort	abt	0b10111
Undefined	und	0b11011
System	sys	0b11111

- **Mode changes**: Can be made under software control, or can be caused by external interrupts or exceptions.
- **User mode**: Most application programs run in User mode, which cannot access protected system resources or to change mode, other than triggering an exception.
- **Privileged modes**: The modes other than User mode are known as privileged modes. They have full access to system resources and can change mode freely.  
Five of them are known as exception modes: FIQ, IRQ, Supervisor, Abort, Undefined. These are entered when specific exceptions occur. Each of them has some additional registers to avoid corrupting User mode state when the exception occurs.
- **System mode**: System mode is one of the privileged modes, and it shares the same register set as the User mode. It has full access to the system's privileged resources.

Switching between these modes is generally controlled by the ARM processor itself in response to certain events (like interrupts). However, software can also change modes explicitly, usually done by system-level code. This is achieved by modifying specific bits in the CPSR.

## Exception level

ARMv7 name	AArch64/ARMv8 name	Remarks	Normal (non-secure) world	Secure world
Monitor mode	EL3	highest exception level, mostly for firmware	EL3 (TrustZone) Monitor	
HYP mode	EL2	exception level for hypervisors like Xen (or parts of KVM)	EL2 Virtual Machine Monitor (VMM) or Hypervisor	AArch64: separate privilege levels AArch32: same privilege level
SVC mode	EL1	the Linux kernel is running in this	EL1 Guest Operating System1 Guest Operating System2	Secure World OS
USR mode	EL0	for unprivileged userland	EL0 App1 App2	Trusted App1 Trusted App2

Exception Levels (ELs): ARMv8 replaces the above modes with a hierarchy of exception levels:

- **EL0**: The least privileged level, typically for application code (similar to User mode in ARMv7).
- **EL1**: For operating system kernel code, similar to the Supervisor mode.
- **EL2**: A new level, typically used by hypervisors in virtualized environments.
- **EL3**: The most privileged level, used for secure boot and trusted execution environments, like TrustZone.

## Registers

# Registers

Modes						
		Privileged modes				
		Exception modes				
User	System	Supervisor	Abort	Undefined	Interrupt	Fast interrupt
R0	R0	R0	R0	R0	R0	R0
R1	R1	R1	R1	R1	R1	R1
R2	R2	R2	R2	R2	R2	R2
R3	R3	R3	R3	R3	R3	R3
R4	R4	R4	R4	R4	R4	R4
R5	R5	R5	R5	R5	R5	R5
R6	R6	R6	R6	R6	R6	R6
R7	R7	R7	R7	R7	R7	R7
R8	R8	R8	R8	R8	R8	R8_fiq
R9	R9	R9	R9	R9	R9	R9_fiq
R10	R10	R10	R10	R10	R10	R10_fiq
R11	R11	R11	R11	R11	R11	R11_fiq
R12	R12	R12	R12	R12	R12	R12_fiq
R13	R13	R13_svc	R13_abt	R13_und	R13_irq	R13_fiq
R14	R14	R14_svc	R14_abt	R14_und	R14_irq	R14_fiq
PC	PC	PC	PC	PC	PC	PC

CPSR	CPSR	CPSR	CPSR	CPSR	CPSR	CPSR
SPSR_svc	SPSR_abt	SPSR_und	SPSR_irq	SPSR_fiq		

 indicates that the normal register used by User or System mode has been replaced by an alternative register specific to the exception mode

- Total of 37 registers(Sum of the registers in the picture above is 37)
  - 31 general-purpose 32-bit registers, including a program counter. R0 ~ R15 visible at any time.  
(The PC is no longer treated as a general-purpose register in ARMv8 or later)
  - 6 status registers

## General-purpose registers

- **Unbanked Registers (R0 to R7 and R15):** Each of them refers to the same 32-bit physical register in all processor modes. They are completely general-purpose registers, with no special uses implied by the architecture
- **Banked Registers (R8 to R14):** Different physical registers depending on the processor mode. A specific name is used to point to a particular physical register.
  - Almost all instructions allow the banked registers to be used wherever a general-purpose register is allowed.
  - Some of the banked registers are unique to the mode, while others are shared (or overlapped) with other modes:
    - Registers R8 to R12 have two banked physical registers each. One is used in all processor modes other than FIQ mode, and the other is used in FIQ mode.
    - Registers R13 and R14 have six banked physical registers each. One is used in User and System modes, and each of the remaining five is used in one of the five exception modes.

## Special Uses of these registers

- R13 (Stack Pointer): Used for stack operations.
- R14 (Link Register): Holds return address after a subroutine call. This is more efficient than using a call stack for every function call, which INTEL uses.
- R15 (Program Counter)(deprecated): R15 can be used in place of other general-purpose registers for certain special-case effects.

By default, R15 operates as a program counter, used for storing the address of the next's next instruction. Storing the next's next instruction is due to the pipeline, where instructions are pre-fetched.

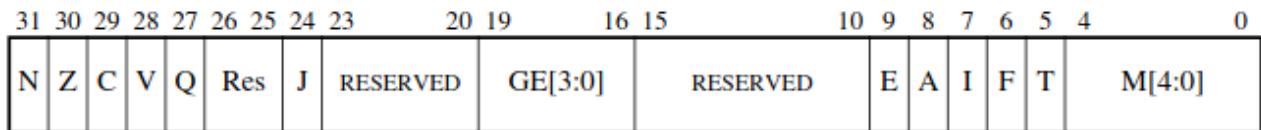
## Program Status Registers

- **Current Program Status Register(CPSR):** Holds processor status, condition code flags, interrupt disable bits, processor mode, etc.
- **Saved Program Status Register(SPSR):** Holds the CPSR of the task before an exception occurred. Each exception mode has a SPSR.

**Table A1-1 Status register summary**

<b>Field</b>	<b>Description</b>	<b>Architecture</b>
N Z C V	Condition code flags	All
J	Jazelle state flag	5TEJ and above
GE[3:0]	SIMD condition flags	6
E	Endian Load/Store	6
A	Inprecise Abort Mask	6
I	IRQ Interrupt Mask	All
F	FIQ Interrupt Mask	All
T	Thumb state flag	4T and above
Mode[4:0]	Processor mode	All

### Types of PSR bits



### Permissions

- **Reserved bits:** Reserved for future expansion. Implementations must read these bits as 0 and ignore writes to them.
- **User-writable bits(N, Z, C, V, Q, GE[3:0], E):** Can be written from any mode.
- **Privileged bits(A, I, F, and M[4:0]):** Can be written from any privileged mode. Writes to privileged bits in User mode are ignored.
- **Execution state bits(J, T):** Can be written from any privileged mode. Writes to execution state bits in User mode are ignored.

### The condition code flags

The N, Z, C, and V (Negative, Zero, Carry and oVerflow) bits are collectively known as the condition code flags, often referred to as flags.

- The condition code flags in the CPSR can be tested by most instructions to determine whether the instruction is to be executed.(if else)
- They are usually modified by the execution of some arithmetic, logical, move instruction, or comparison instruction (CMN, CMP, TEQ or TST).

### The Q flag

bit[27] of the CPSR is known as the Q flag and is used to indicate whether overflow and/or saturation has occurred in some DSP-oriented instructions.

## The GE[3:0] bits

The SIMD instructions use bits[19:16] as Greater than or Equal (GE) flags for individual bytes or halfwords of the result.

You can use these flags to control a later SEL instruction

## The E bit

Controls load and store endianness for data handling.

## The interrupt disable bits

- **A bit:** Disables imprecise data aborts when it is set. This is available only in ARMv6 and above. In earlier versions, bit[8] of CPSR and SPSRs must be treated as a reserved bit, as described in Types of PSR bits on page A2-11. I bit Disables IRQ interrupts when it is set. F bit Disables FIQ interrupts when it is set.

## The mode bits

M[4:0] are the mode bits. These determine the mode in which the processor operates.

**Table A2-2 The mode bits**

M[4:0]	Mode	Accessible registers
0b10000	User	PC, R14 to R0, CPSR
0b10001	FIQ	PC, R14_fiq to R8_fiq, R7 to R0, CPSR, SPSR_fiq
0b10010	IRQ	PC, R14_irq, R13_irq, R12 to R0, CPSR, SPSR_irq
0b10011	Supervisor	PC, R14_svc, R13_svc, R12 to R0, CPSR, SPSR_svc
0b10111	Abort	PC, R14_abt, R13_abt, R12 to R0, CPSR, SPSR_abt
0b11011	Undefined	PC, R14_und, R13_und, R12 to R0, CPSR, SPSR_und
0b11111	System	PC, R14 to R0, CPSR (ARMv4 and above)

## The T and J bits

**Table A2-3 The T and J bits**

J	T	Instruction set
0	0	ARM
0	1	Thumb
1	0	Jazelle
1	1	RESERVED

The T and J bits select the current instruction set, as shown in Table A2-3.

## Other bits

Other bits in the program status registers are reserved for future expansion.

In general, programmers must take care to write code in such a way that these bits are never modified. Failure to do this might result in code that has unexpected side effects on future versions of the architecture.

## Exceptions

Exceptions are generated by internal and external sources to cause the processor to handle an event, such as an externally generated interrupt.

More than one exception can arise at the same time.

All exception modes have replacement banked registers. When an exception occurs, standard registers are replaced with registers specific to the exception mode.

## Exception process

- When an exception occurs, execution is forced from a fixed memory address corresponding to the type of exception. These fixed addresses are called the **exception vectors**. There is a separate vector location for each exception.
- The banked versions of R14 and the SPSR are used to save state so that the original program can be resumed when the exception routine has completed.
- To return after handling the exception, the SPSR is moved into the CPSR, and R14 is moved to the PC.

## Types of exceptions

The ARM architecture supports seven types of exception. Table A2-4 lists the types of exception and the processor mode that is used to process each type.



**Reset**

When the Reset input is asserted on the processor, the ARM processor immediately stops execution of the current instruction.

**Undefined Instruction exception**

Occurs when the processor encounters an instruction that is either **not defined** in the ARM instruction set or is **not valid** for the current state of the processor

**Software Interrupt exception**

The Software Interrupt instruction (SWI) enters Supervisor mode to request a particular supervisor function. (OS system call)

**Prefetch Abort (Instruction fetch memory abort)**

- **Precondition to occur:** A memory abort is signaled by the memory system. Attempting to fetch an instruction from a memory location that it cannot access correctly marks the data as invalid.
- **Occurs when:** A Prefetch Abort exception is generated if the processor tries to execute the invalid instruction.
- **Does not occur when:** If the instruction is not executed (for example, as a result of a branch being taken while it is in the pipeline), no Prefetch Abort occurs.

**Data Abort (Data access memory abort)**

A data abort occurs when a problem arises during a data access operation, such as reading from or writing to memory.

**Interrupt request (IRQ) exception**

The IRQ exception is generated externally by asserting the IRQ input on the processor.

**Fast interrupt request (FIQ) exception**

The FIQ exception is generated externally by asserting the FIQ input on the processor. FIQ is designed to support a data transfer or channel process.

- Why fast interrupt is faster
  - **Dedicated Registers:** FIQ has additional banked registers(private) to remove the need for register saving in such applications, therefore minimizing the overhead of context switching.
  - **Shorter vector:** There are fewer instructions to process before the ISR code is reached.
  - **Higher Priority:** FIQ has a higher priority than IRQ

## Exception priorities

**Table A2-4 Exception processing modes**

Exception type	Mode	VE <sup>a</sup>	Normal address	High vector address
Reset	Supervisor		0x00000000	0xFFFF0000
Undefined instructions	Undefined		0x00000004	0xFFFF0004
Software interrupt (SWI)	Supervisor		0x00000008	0xFFFF0008
Prefetch Abort (instruction fetch memory abort)	Abort		0x0000000C	0xFFFF000C
Data Abort (data access memory abort)	Abort		0x00000010	0xFFFF0010
IRQ (interrupt)	IRQ	0	0x00000018	0xFFFF0018
		1	IMPLEMENTATION DEFINED	
FIQ (fast interrupt)	FIQ	0	0x0000001C	0xFFFF001C
		1	IMPLEMENTATION DEFINED	

a. VE = vectored interrupt enable (CP15 control); RAZ when not implemented.

## Endian

Endianness is the order of bytes of digital data.

ARMv6 supports both big-endian and little-endian operation  
(Least Significant bit: the bit that has the lowest value)

- In little-endian mode, the least significant bit is stored at the smallest address. (most common)
- In big-endian mode, the most significant bit is stored at the smallest address.



## Unaligned access support

- **Aligned Access:** An access is aligned if the memory address being accessed is a multiple of the size of the data type. For example, accessing a 4-byte integer at memory addresses 0, 4, 8, etc., is considered aligned.  
Aligned memory accesses are typically faster because they align with the natural boundaries of memory buses and caches.
- **Unaligned Access:** An access is unaligned if the memory address is not a multiple of the size of the data type. For instance, accessing a 4-byte integer at memory addresses 2, 6, 10, etc., would be unaligned.

Changes with ARMv6:

Traditionally, ARM expects memory accesses to be aligned.  
ARMv6 introduced support for unaligned word and halfword data access.

## Synchronization primitives

### LDREX (Load-Exclusive)

- When a thread executes an LDREX instruction, it reads data from a memory location and the processor marks this location as having been accessed exclusively by this thread.
- However, this 'exclusive' mark doesn't block other threads or cores from accessing the same memory location. They can still read from and write to this location.

### STREX (Store-Exclusive)

- STREX checks if the memory location still has its exclusive access status. If no other thread has written to this location since the LDREX was executed, STREX considers the operation successful and writes the new value.
- If, however, another thread has written to the location, the STREX operation will fail, returning a non-zero value.

Typical Usage in a Loop:

In practice, the thread often repeatedly reads (LDREX) and tries to write (STREX) in a loop until STREX reports success (indicating no other writes occurred in the meantime).

## Jazelle extension

- **Background:** Jazelle was introduced to mitigate the overheads of JVM by allowing for the native execution of Java bytecode.

- **Operation:** When in Jazelle mode, the processor can directly execute Java bytecode instructions. This is achieved by mapping Java bytecodes to equivalent ARM processor instructions or sequences of instructions.
- **Decline of Jazelle:** The advancements in JIT have largely overshadowed the need for direct bytecode execution.

### Saturated integer arithmetic

Saturated arithmetic prevents the overflow/underflow. Instead, if an operation results in a value outside the representable range, it is set to the closest representable value (the maximum or minimum value of the data type).

- **For instance,** in 8-bit saturated arithmetic, if adding two numbers results in a value greater than 255, the result is set to 255. Similarly, if subtraction would result in a negative value, the result is set to 0.

## 3. Memory.

### Memory

#### About the VMSA

- **Virtual Memory System:** VMSA involves dynamically allocating memory and mapping it to physical addresses via a Memory Management Unit (MMU).
- **Translation Table Walk:** This is the process of converting virtual addresses to physical ones, searching page tables.
- **Translation Lookaside Buffers (TLBs):** TLBs store the results of recent translation table lookups.(not the entire contents) If a virtual address translation is in the TLB, the MMU can use it directly, avoiding a full table walk.

#### Enhancements in ARMv6 (VMSA6)

VMSA has been improved in ARMv6 to avoid the need for TLB invalidation on a context switch, thus improving performance.

The key enhancements include:

- Global mappings for applications, avoiding the need for TLB flushes on most context switches.
- New memory types for efficient memory management.
- Enhanced memory properties in TLB entries for better access control.

#### Memory Access Sequence

When the ARM CPU generates a memory access, the MMU performs a lookup for a modified virtual address in a TLB. This includes checking the current ASID in ARMv6 implementations.

- **Modified virtual address:** refers to a combination of the 32-bit virtual address along with additional bits for the Application Space Identifier (ASID), which provides a way to separate the address spaces of different processes.
- **ASID (Address Space Identifier):** Uniquely identifies a process's virtual memory space. It is used to distinguish between memory pages with the same virtual address but used by different tasks.
- Instruction fetches use the *instruction TLB* and data accesses use the *data TLB*.
- If no appropriate TLB entry is found, a translation table walk is performed by hardware.

## Note

- Modified virtual address translations are globally mapped from ARMv6, considering the 32-bit modified virtual address plus ASID values when non-global address is accessed.
- The FCSE mechanism is deprecated in ARMv6. Concurrent use of FCSE and ASID leads to unpredictable behavior.

## TLB Match Process

Each TLB entry contains a modified virtual address, a page size, a physical address, and memory properties. It's associated with an application space identifier (ASID) or marked as global.

- A match occurs if the higher order bits of the modified virtual address match and the ASID is the same.
- The behavior of a TLB is unpredictable if multiple entries match at any time.

## Enabling and disabling the MMU

Controlling the MMU:

- The MMU is enabled and disabled by setting or clearing the M bit (bit[0]) in register 1 of the System Control coprocessor (CP15).
- It's disabled by default upon reset.

Memory Access Behavior When MMU is Disabled:

### Prior knowledge

**Strongly ordered memory:** Memory accesses to Strongly Ordered memory regions are strictly ordered. This means that any access (read or write) to a Strongly Ordered memory location must be completed before any subsequent access can begin. This order is maintained across all processors in a multi-core system.

### Data Accesses:

- Treated as uncacheable and strongly ordered.
- Unexpected data cache hit behavior varies depending on implementation.

**Instruction Accesses:**

Behavior depends on cache architecture:

- Harvard cache: Cacheable or non-cacheable based on I bit (bit[12]) of CP15 register 1.
- Unified cache: Always non-cacheable.

**Explicit Accesses:**

- Strongly ordered.
- W bit (bit[3], write buffer enable) of CP15 register 1 is ignored.

**Memory Access Permissions and Aborts:**

- No permission checks are performed.
- No aborts are generated by the MMU.

**Address Mapping:**

- Flat address mapping: Physical address = modified virtual address.

**FCSE PID:**

- Should be zero (SBZ) when MMU is disabled.
- Clearing FCSE PID is recommended before disabling MMU.

**Cache and TLB Operations:**

- Cache CP15 operations work regardless of MMU state, but use flat mapping when MMU is disabled.
- CP15 TLB invalidate operations work regardless of MMU state.

**Other Operations:**

- Instruction and data prefetch operations work normally.
- Accesses to TCMs (Tightly Coupled Memories) work normally if TCM is enabled.

**Prerequisites for Enabling MMU:**

- **CP15 Register Programming:** Set up relevant CP15 registers, including translation tables.
- **Instruction Cache Handling:**
  - Disable and invalidate instruction cache before enabling MMU.
  - Re-enable instruction cache along with MMU.

## Memory access control

Access to a memory region is controlled by the access permission and domain bits in the TLB entry. APX and XN (execute never) bits have been added in VMSAv6.

### Access permissions

- **Permission fault:** If an access is made to an area of memory without the required permissions, a Permission Fault is raised.
- **Access permissions:** The access permissions are determined by a combination of the AP and APX bits in the page table.

### Domains(deprecated)

- A domain is a named collection of memory regions. The ARM architecture supports up to 16 domains.
- Each page table entry and TLB entry contains a field that specifies the domain the entry belongs to.
- Access to each domain is controlled by a two-bit field in the Domain Access Control Register (DACR).

#### DACR:

- The DACR is a register that holds 16 two-bit fields, one for each domain.
- These fields can be set to one of three values:
  - b'00: No access - Any attempt to access memory in that domain will generate a domain fault.
  - b'01: Client - Accesses are allowed, but they must follow the access permissions set in the TLB entries for that domain.
  - b'11: Manager - Accesses are allowed without checking the TLB entries. This gives manager programs full control over the domain, but also bypasses security checks.

## Memory region attributes

### Prior to VMSAv6(Problem)

- Only two bits were used for memory region attributes: C (cacheable) and B (bufferable).
- **Incompatible:** The exact usage of these bits was implementation defined, which meant different ARM processors could behave differently, even if they used the same memory region attributes.

### VMSAv6 and Later

- VMSAv6 introduced a more formal memory model, with additional bit fields and definitions for memory region attributes.
- These attributes provide more consistent behaviors across different ARM processors.

## Key Memory Region Attributes

- TEX (Type Extension): This field provides additional information about the memory region type, such as whether it is normal memory, device memory, or strongly ordered memory.
- C (Cacheable): This bit controls whether the memory region can be cached.
- B (Bufferable): This bit controls whether writes to the memory region can be buffered in the write buffer.
- S (Shared): This bit (only present in certain page tables) controls whether the memory region is shared between multiple processors.

## Page Table Formats

- The TEX, C, and B bits are encoded in the page table entries.
- Table B4-3 in the ARM Architecture Reference Manual shows the mapping of these bits to different memory region types.

**Table B4-3 CB + TEX Encodings**

<b>TEX</b>	<b>C</b>	<b>B</b>	<b>Description</b>	<b>Memory type</b>	<b>Page shareable</b>
0b000	0	0	Strongly ordered	Strongly ordered	Shareable
0b000	0	1	Shared Device	Device	Shareable
0b000	1	0	Outer and inner write through, no write allocate	Normal	S
0b000	1	1	Outer and inner write back, no write allocate	Normal	S
0b001	0	0	Outer and inner non-cacheable	Normal	S
0b001	0	1	RESERVED	-	-
0b001	1	0	IMPLEMENTATION DEFINED	IMPLEMENTATION DEFINED	IMPLEMENTATION DEFINED
0b001	1	1	Outer and inner write back, write allocate	Normal	S
0b010	0	0	Non-shared device	Device	Not shareable
0b010	0	1	RESERVED	-	-
0b010	1	X	RESERVED	-	-
0b011	X	X	RESERVED	-	-
0b1BB	A	A	Cached memory BB = outer policy, AA = inner policy	Normal	S

## Aborts

Aborts are exceptions caused by memory access issues.

- **MMU fault:** Triggered by the MMU when it detects a memory access violation.
- **Debug abort:** Occurs when the processor's debug mode is active, and a breakpoint or watchpoint is hit.
- **External abort:** Caused by illegal or faulting memory accesses from the external memory system.

Faults(aborts) are recorded using the **Fault Address** and **Fault Status registers**.

## MMU Faults

The MMU can generate four types of faults:

- **Alignment fault:** Occurs when memory access is not aligned correctly.
- **Translation fault:** Happens when there is an error translating a virtual memory address to a physical one.
- **Domain fault:** Triggered when access to a domain is not allowed or improperly configured.
- **Permission fault:** Arises when there is an attempt to access memory with improper permissions.

### Fault-checking Sequence (Figure B4-2)

The MMU follows a sequence to check for faults to determine if a fault should be reported.

### Translation and Domain Faults

- **Section translation fault:** Occurs when the first-level descriptor is invalid.
- **Section domain fault:** Occurs when the domain of the first-level descriptor is checked and found to be incorrect.
- **Page translation fault:** Occurs when the second-level descriptor is invalid.
- **Page domain fault:** Similar to section domain faults but at the second descriptor level.

## Debug Events

- **Precise aborts:** When the exact address of the failing instruction is known.
- **Imprecise aborts:** When the exact address is not known due to subsequent instructions executing(pipeline) before the abort is processed.

## External Aborts

External aborts are errors from the external memory system and can be precise or imprecise, affecting the Fault Address register and Fault Status register updates.

## Parity Error Reporting

Parity errors can be precise or imprecise, with specific fault status codes assigned for reporting these errors. These are implementation-defined.

## Fault Address and Fault Status registers

Registers:

Prior to VMSAv6, the architecture supported a single Fault Address Register (FAR) and Fault Status Register (FSR).

VMSAv6 requires four registers:

- **Instruction Fault Status Register (IFSR):** Updated on Prefetch Aborts.
- **Data Fault Status Register (DFSR):** Updated on Data Aborts.
- **Fault Address Register (FAR):** Contains the faulting address for precise exceptions.
- **Watchpoint Fault Address Register (WFAR):** Updated on a watchpoint access that triggers a Data Abort.

Notes:

- IFSR and DFSR are updated on Data Aborts due to instruction cache maintenance operations.
- A fifth fault status bit (bit[10]) for IFSR and DFSR is implementation-specific. DFSR also includes a write flag (bit[11]).
- Precise Data Aborts prompt immediate CPU action, updating DFSR with Fault Status and domain number. FAR records the address causing the abort.
- Instruction fetches that cause aborts update IFSR but not FAR unless the instruction is executed.
- For Precise Data Aborts, the Fault Address is determined from R14 at the time the Prefetch Abort exception is encountered.

## Notes for Fault Status Register Encodings Table (B4.6.1)

- Prior to VMSAv6, FS[3:0] values were implementation-defined.
- In VMSAv6, Domain information for Data Accesses is obtained via TLB lookup. For Prefetch Aborts, faulting address and domain are determined from the TLB.
- All Data Aborts affect DFSR; all Instruction Aborts affect IFSR.
- Data Aborts not from external translation update FAR with the aborting address. If from external translation, FAR does not contain the aborting address.
- Translation aborts during data cache maintenance operations update DFSR with the reason and FAR with the faulting address.
- Precise Aborts during instruction cache maintenance are indicated in DFSR, with the modified virtual address reflected in IFSR.

## Hardware page table translation

**MMU Function:** Translates memory accesses based on sections or pages.

Memory Sections:

- *Supersections* (optional, 16MB)
- *Sections* (1MB)

Supported page sizes:

- *Large pages* (64KB)
- *Small pages* (4KB)
- *Tiny pages* (1KB, not in VMSAv6)

Translation Tables: Two-level structure in main memory for translating virtual addresses to physical addresses.

- **First-level table:** Contains section and supersection translations, and pointers to second-level tables.
- **Second-level tables:** Handle translations for pages.

First-level Fetch

- **Address Translation:** Bits[31:14] of the Translation Table Base Register are used with modified virtual addresses to fetch descriptors or pointers to second-level tables.

Etc

- **On TLB Miss:** When a TLB miss occurs, the Translation Table Base Register (in CP15 register 2) holds the base address of the first-level table.
- **Endian Configuration:** The EE-bit in the System Control coprocessor determines endianness during table lookups.

## Page Table Translation in VMSAv6

VMSAv6 supports two page table formats:

1. **Backward-Compatible Format:** This format supports sub-page access permissions and has been extended to support extended region types.
2. **New Format:** This format does not support sub-page access permissions but includes support for the following features:
  - Extended region types
  - Global and process-specific pages

- More access permissions
- Marking of shared and non-shared regions
- Marking of execute-never regions.

### First-Level Descriptors(Entries in the first-level translation table)

- When a virtual address is translated to a physical address, the MMU first looks at the first-level translation table.
- The first-level descriptor can either directly point to a large section of physical memory (like a 1MB section), or it can point to a second-level translation table that handles smaller blocks of memory.

The first-level translation table contains entries that can be one of several types:

- **Supersection Descriptor:** Similar to a section descriptor, a supersection descriptor maps a larger block of memory (typically 16MB) and also does not point to a second-level table.
- **Section Descriptor:** This type of entry maps a full 1MB section of virtual address space directly to physical memory. It does not point to a second-level table but instead provides the physical base address of the section along with access permissions and other attributes.
- **Coarse Page Table Descriptor:** This entry does point to a second-level table, specifically a **coarse second-level page table**. Each entry in the coarse page table then maps a smaller portion of memory (typically 4KB pages).
- **Fine Page Table Descriptor(OBSOLETE):** This is similar to the coarse page table descriptor but allows for an even finer granularity of mapping (typically **1KB** pages) and thus points to a **fine second-level page table** with more entries than the coarse table.

### Second-Level Descriptor(Entries in the second-level translation table)

- If the first-level descriptor points to a second-level translation table, the MMU then uses this table for further translation.
- Second-level descriptors provide a finer granularity of memory management, allowing for smaller sections of memory to be individually controlled, such as 64KB large pages or 4KB small pages.
- These descriptors are used to define more specific attributes of the memory region, such as access permissions, cache behavior, and whether the memory is buffered.

## CP15 Registers (deprecated in ARMv8)

- **Memory management:** This includes the control of the Memory Management Unit (MMU), Translation Lookaside Buffers (TLB), and cache control for instruction and data caches.
- **Protection:** CP15 registers define and control the different protection regions in the memory to prevent unauthorized access.
- **System identification:** These registers can hold information about the processor, such as the manufacturer ID, CPU ID, and version numbers.
- **Performance monitoring:** Certain CP15 registers are used to monitor the performance of the CPU, including counting cache misses, instruction execution, and other performance metrics.
- **Configuration:** CP15 registers allow the configuration of other system settings, like the endianness of data processing, interrupt handling, and so forth.

## Protected Memory System Architecture (PMSA)

### Overview of PMSA

- **Functionality:** Controls access to memory regions.(hardware based)
- **Target Devices:** Primarily used in embedded systems and low-power devices.
- **Memory Protection Unit (MPU):** Central to PMSA, the MPU controls access rights.

### Memory protection

- **Protection Attributes:** Each memory segment is assigned specific attributes (read/write/execute permissions).
- **Fault Handling:** If a task accesses a memory region it is not permitted to without proper permissions, a fault is generated.
- **Isolation of Faults:** Limits the impact of faults to specific regions.

### Operational Modes

- **Privileged Mode:** Full access to all resources, usually for the operating system.
- **Unprivileged Mode:** Restricted access, typically for application code.

### Advantages and Limitations

- **Advantages:** Enhanced security, fine-grained control over memory, and fault isolation.
- **Limitations:** Complexity in configuration, overhead in terms of memory and processing.

## BUS 양영식

### APB bus 양영식

## APB

The APB (Advanced Peripheral Bus) is part of the AMBA (Advanced Microcontroller Bus Architecture) suite of protocols.

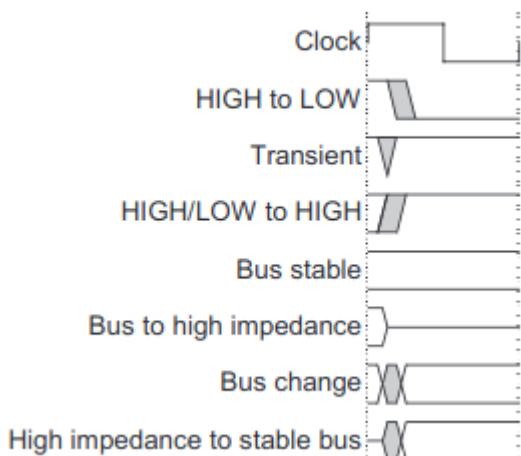
AMBA is a widely used interconnection standard for connecting functional blocks (like CPUs, memory units, and peripherals).

- **Purpose:** APB is designed for low-bandwidth control accesses, for example, peripheral device control. Its primary use is to connect simple peripherals to the primary bus, which could be an AHB (Advanced High-performance Bus) or AXI (Advanced eXtensible Interface) bus in the AMBA standard.

- **Simplicity and Efficiency:** The APB is simpler compared to other buses in the AMBA family. It is optimized for minimal power consumption and reduced complexity of peripherals. This makes it ideal for simple, lower-speed tasks where high throughput is not a critical requirement.
- **Use Cases:** APB is commonly used for interfacing with low-speed peripherals like keyboards, mouse devices, UARTs, SPI (Serial Peripheral Interface) controllers (control signals don't require high speed).

## Prior knowledge

### Timing diagram conventions



- **Bus change:** This indicates that multiple bits on the bus are changing simultaneously aside from just a single binary digit changing
- **Transient:** This is a temporary, intermediate state during the transition of a signal from one stable state to another. It's a period where the signal is neither high nor low
- **Bus to high impedance:** This shows the bus moving to an electrical disconnection. This state is used when multiple devices can drive the bus, but only one should do so at a time to avoid conflicts.

## APB signal descriptions

### 4.1 AMBA 3 APB signals

Table 4-1 lists the APB signal descriptions.

**Table 4-1 APB signal descriptions**

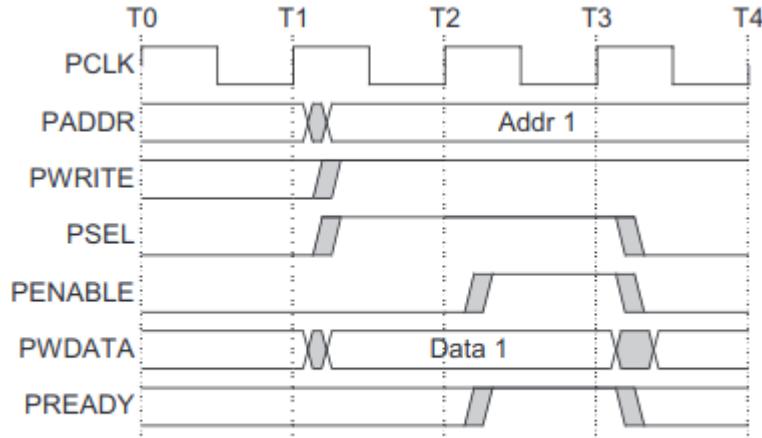
Signal	Source	Description
<b>PCLK</b>	Clock source	Clock. The rising edge of <b>PCLK</b> times all transfers on the APB.
<b>PRESETn</b>	System bus equivalent	Reset. The APB reset signal is active LOW. This signal is normally connected directly to the system bus reset signal.
<b>PADDR</b>	APB bridge	Address. This is the APB address bus. It can be up to 32 bits wide and is driven by the peripheral bus bridge unit.
<b>PSELx</b>	APB bridge	Select. The APB bridge unit generates this signal to each peripheral bus slave. It indicates that the slave device is selected and that a data transfer is required. There is a <b>PSELx</b> signal for each slave.
<b>PENABLE</b>	APB bridge	Enable. This signal indicates the second and subsequent cycles of an APB transfer.
<b>PWRITE</b>	APB bridge	Direction. This signal indicates an APB write access when HIGH and an APB read access when LOW.
<b>PWDATA</b>	APB bridge	Write data. This bus is driven by the peripheral bus bridge unit during write cycles when <b>PWRITE</b> is HIGH. This bus can be up to 32 bits wide.
<b>PREADY</b>	Slave interface	Ready. The slave uses this signal to extend an APB transfer.
<b>PRDATA</b>	Slave interface	Read Data. The selected slave drives this bus during read cycles when <b>PWRITE</b> is LOW. This bus can be up to 32-bits wide.
<b>PSLVERR</b>	Slave interface	This signal indicates a transfer failure. APB peripherals are not required to support the <b>PSLVERR</b> pin. This is true for both existing and new APB peripheral designs. Where a peripheral does not include this pin then the appropriate input to the APB bridge is tied LOW.

### 2.1 Write Transfers

Write transfers can be categorized into two types:

- With no wait states
- With wait states

### 2.1.1 With no wait states

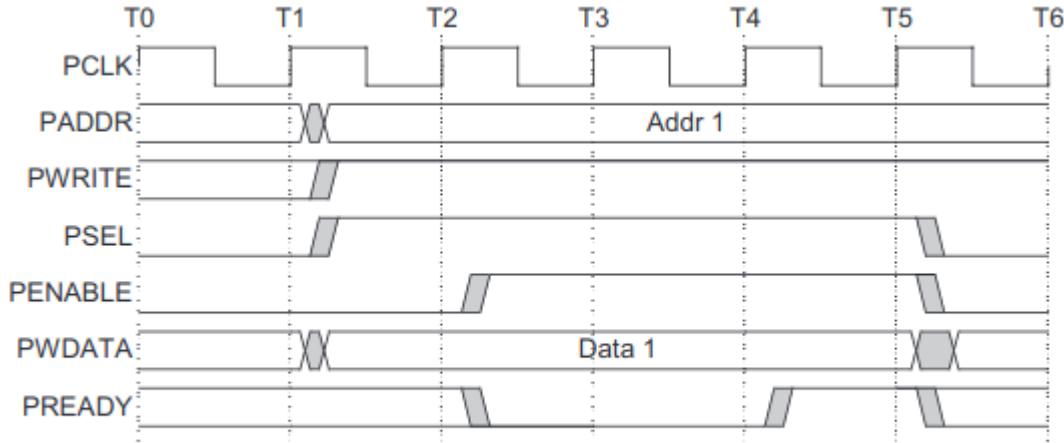


**Figure 2-1 Write transfer with no wait states**

- **At T0:** The process is idle, waiting for the next clock cycle.
- **T1:** The address ( PADDR ) and write data ( PWDATA ) become valid, and the write signal ( PWRITE ) and select signal ( PSEL ) are asserted.
- **T2-T3:** The PENABLE signal is asserted, indicating the Access phase is in progress, during which the address and data remain valid.
- **T4:** The PENABLE signal is deasserted, marking the end of the transfer. If another transfer is not immediately following, PSEL goes LOW.

The transfer completes at the end of this cycle, with all signals remaining valid throughout the Access phase.

### 2.1.2 With wait states



**Figure 2-2 Write transfer with wait states**

The `PREADY` signal can extend the transfer by driving `PREADY` LOW when `PENABLE` is HIGH.

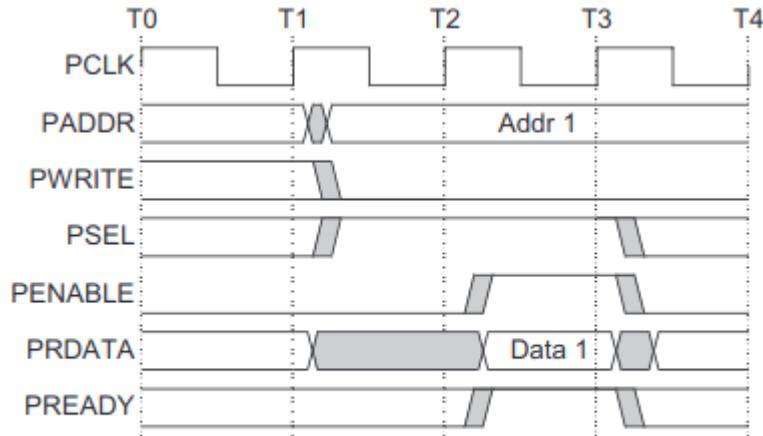
- **T2-T4:** The `PENABLE` signal is asserted, but the `PREADY` signal goes LOW, indicating the wait states.

*Note: It is recommended that the address and write signals remain stable until another access occurs to reduce power consumption.*

## 2.2 Read transfers(same as write)

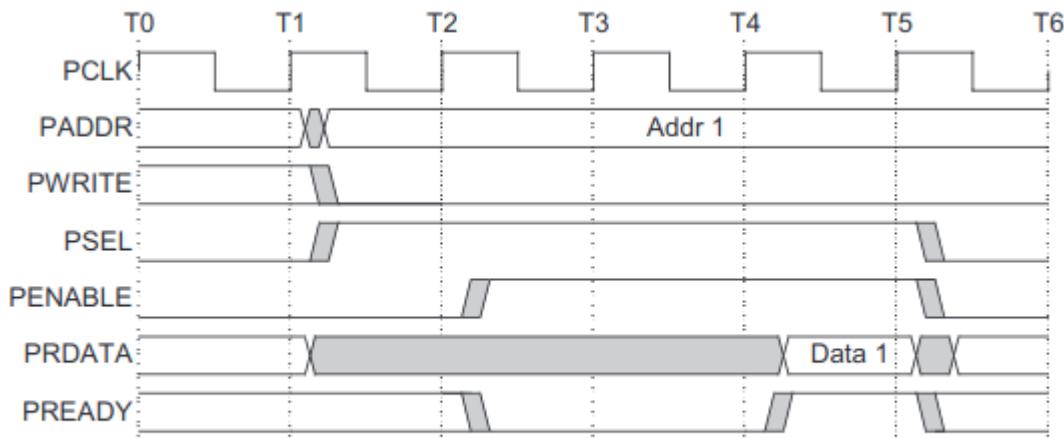
The timing of the signals is similar to the write transfer(as described in `Write transfers` on section 2.1) but is oriented towards reading data from a slave device.

### 2.2.1 With no wait states



**Figure 2-3 Read transfer with no wait states**

### 2.2.2 With wait states



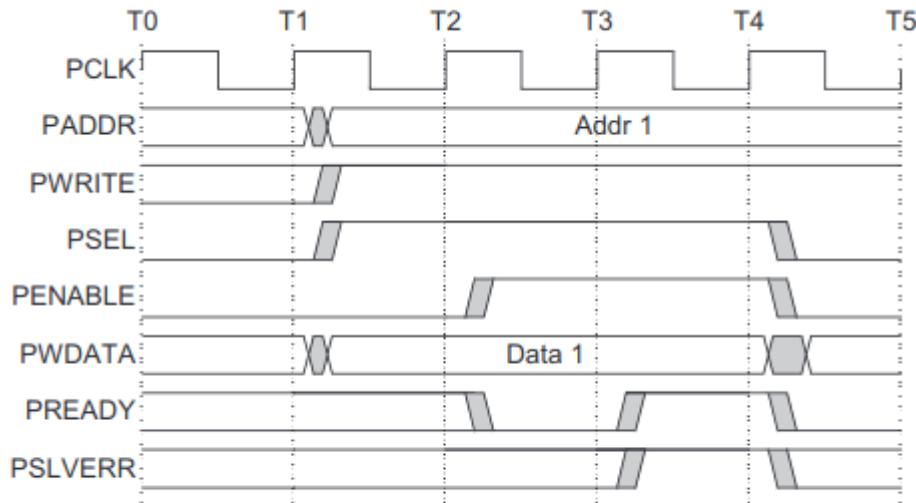
**Figure 2-4 Read transfer with wait states**

## 2.3 Error response

- **PSLVERR**: It's a signal used to indicate an error during an APB transfer.
- **Validity**: PSLVERR is valid only during the last cycle of an APB transfer when PSEL, PENABLE, and PREADY are all HIGH.
- **Recommendation**: It's suggested to drive PSLVERR LOW when not being sampled, i.e., when any of PSEL, PENABLE, or PREADY are LOW.

- **Either behavior:** Whether an error changes the state of the peripheral is peripheral-specific and either behavior is acceptable.

### 2.3.1 Failing transfer



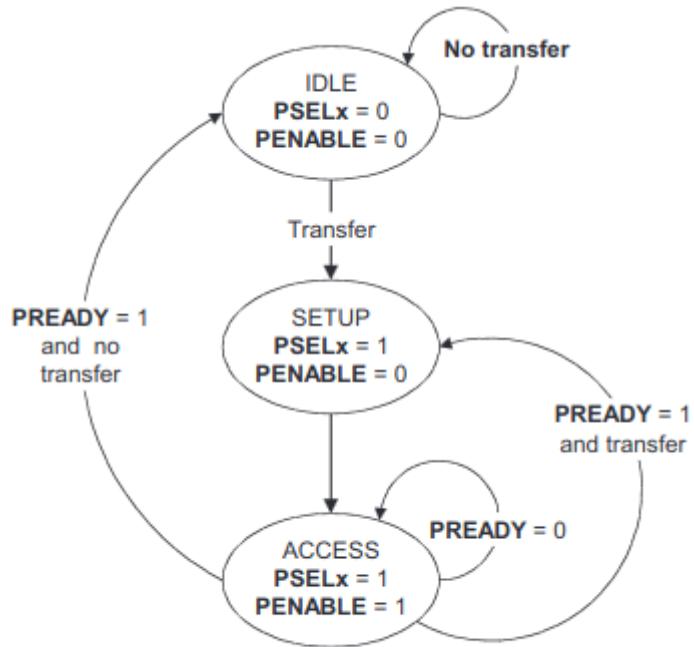
**Figure 2-5 Example failing write transfer**

The high PSLVERR indicates that there has been an error during the transfer.

### 2.3.3 Mapping of PSLVERR

- **AXI to APB:** An APB error is mapped back to RESP/BRESP = SLVERR. This is done by mapping PSLVERR to the AXI signals RRESP[1] for reads and BRESP[1] for writes.
- **AHB to APB:** PSLVERR is mapped back to HRESP = ERROR, achieved by mapping PSLVERR to the AHB signal HRESP[0].

### 3.1 Operating States



**Figure 3-1 State diagram**

This diagram visually represents the transitions between the different states of the APB state machine. It shows:

- **IDLE**: The default state of the APB when no transfers are occurring.
- **To SETUP**: When a transfer is required, the bus moves into the **SETUP** state.
- **SETUP**: The peripheral select signal `PSELx` is asserted, indicating which peripheral is being communicated with.
  - **To ACCESS**: The bus only remains in this state for one clock cycle and moves to the **ACCESS** state on the next clock's rising edge.
- **ACCESS**: The enable signal `PENABLE` is asserted in the **ACCESS** state. Stable signals(address, write, select, and write data) are crucial during the transition from **SETUP** to **ACCESS**. The exit from the **ACCESS** state is dependent on the `PREADY` signal from the peripheral (slave device).
  - **Remaining in ACCESS**: If `PREADY` is low, the bus remains in the **ACCESS** state.
  - **Exit from ACCESS**: If `PREADY` is high, the bus either returns to the **IDLE** state if no more transfers are needed **OR** moves directly to the **SETUP** state if another transfer is pending.

## AXI bus 양영식

### Introduction

AXI is designed to provide a high-performance communication interface between hardware components.

- **Terminology:** A transaction is a sequence of multiple operations performed as a single unit.

### Key features of AXI

- **High Performance:** Supports high-speed data transfer and efficient communication between different parts of a system.
- **Separate Read and Write Channels:** It has independent channels for read and write operations, enhancing throughput and efficiency.
- **Separate Address(control) and Data channels:** It allows address information to be issued ahead of the actual data transfer.
- **Support for Burst Transfers:** AXI allows burst-based transfers, which is efficient for moving large blocks of data.
- **Out-of-Order Transaction:** It allows for transactions to complete in any order, not necessarily the order they were initiated.
- **Flexible and Scalable:** It's adaptable to various types of implementations and can be scaled according to the system requirements.
- **Support for Multiple Masters and Slaves:** AXI can handle communication between multiple master and slave devices, making it suitable for complex SoC (System on Chip) designs.

### Architecture

The AXI protocol is designed to be burst-based and supports high-performance data transfers between master and slave components.

### Channels

The AXI protocol defines five independent channels, each employing a `VALID` and `READY` handshake mechanism to ensure proper data transfers.

- **Master Devices:** Initiate transactions.
- **Slave Devices:** These are the responders to the transactions initiated by the masters.

### Channels

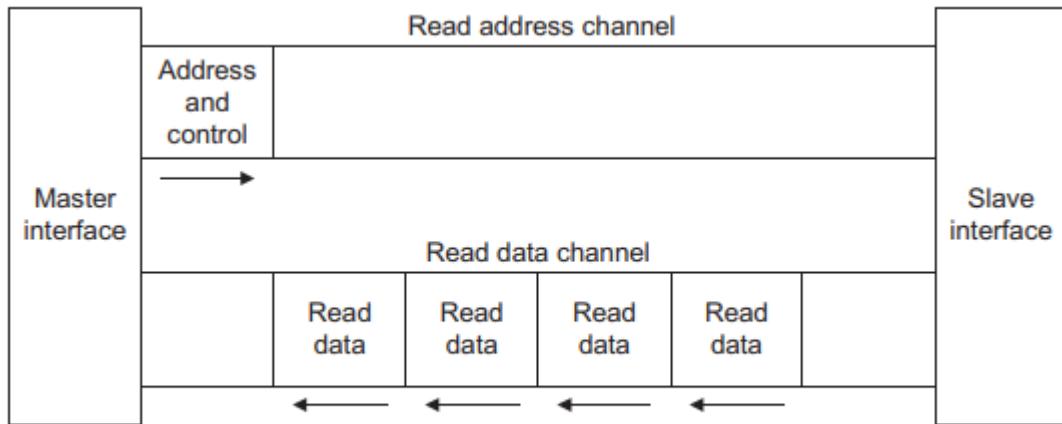
- **Read Address and Write Address Channels:** Read and write transactions each have their own address channel. These carry the address and control information for their respective transactions.
- **Read Data Channel:** This channel conveys both the read data and read response from the slave to the master.
- **Write Data Channel:** This channel carries the write data from the master to the slave.

- **Write Response Channel:** Provides a way for the slave to respond to write transactions. All write transactions use completion signaling, which occurs once for each burst, not for each individual data transfer within the burst.

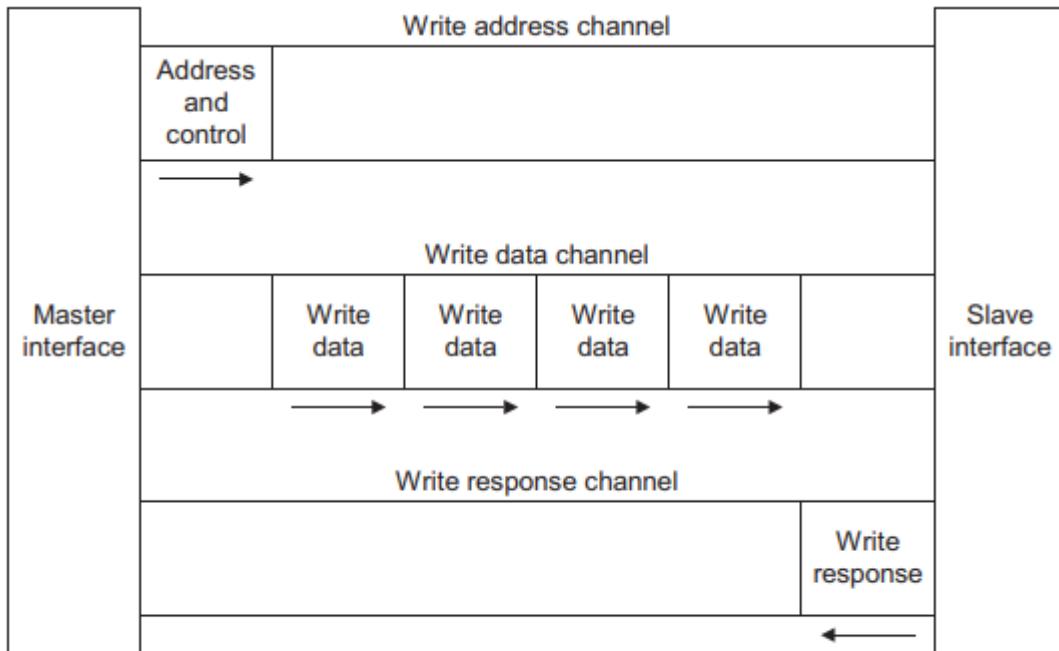
### Separate address and data channel

Shows how a read transaction uses the read address and read data channels.

**Figure 1-1**



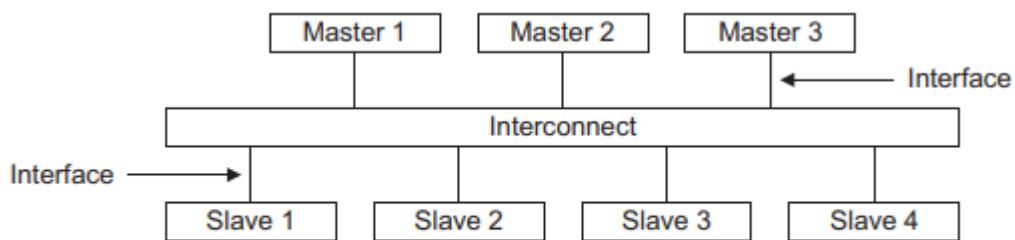
**Figure 1-1 Channel architecture of reads**

**Figure 1-2****Figure 1-2 Channel architecture of writes**

- **Mandatory Relationships:**

- Read data must always follow the address to which the data relates.
- Write response must always follow the last write transfer in the write transaction.

### Interconnect

**Figure 1-3 Interface and interconnect**

- **Interconnect:** The interconnect allows communication between master and slave devices.

## Channel Handshake

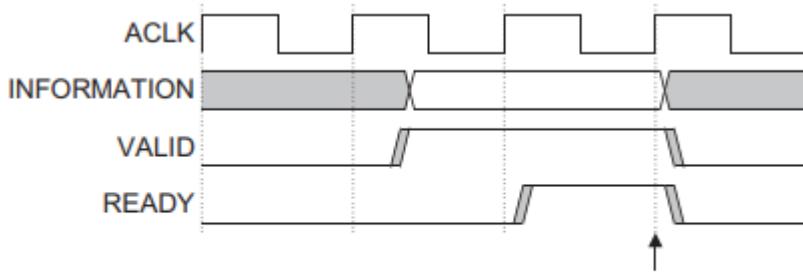
### Handshake Process

It occurs on each burst

- **Purpose:** The handshake process in AXI is used to synchronize data and control information between the master and slave devices.(like TCP handshake)
- **Mechanism:** It uses a two-way VALID/READY flow control system to ensure data transfer occurs at an appropriate rate without loss or error.
- **Signals:**
  - **VALID:** Indicates data or control information is available from the source.
  - **READY:** Indicates the destination can accept the data or control information.
  - A **LAST** signal is also employed to denote the end of a burst of data transfers.
- **Conditions:** Both VALID and READY signals must be high for the data transfer to take place.

### Timing Diagrams

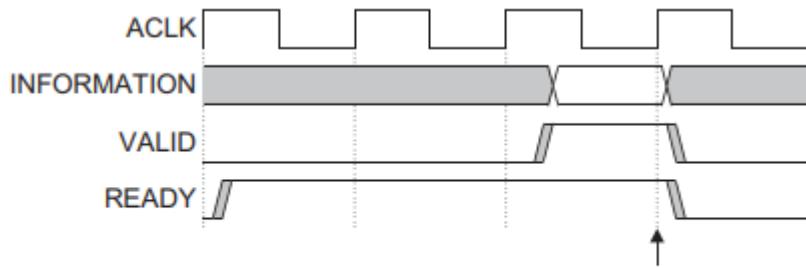
#### VALID before READY Handshake:



**Figure 3-1 VALID before READY handshake**

- The source sets the VALID signal high when data or control information is ready to be sent.
- The destination sets the READY signal high to accept the transfer.
- The transfer takes place when both signals are high.

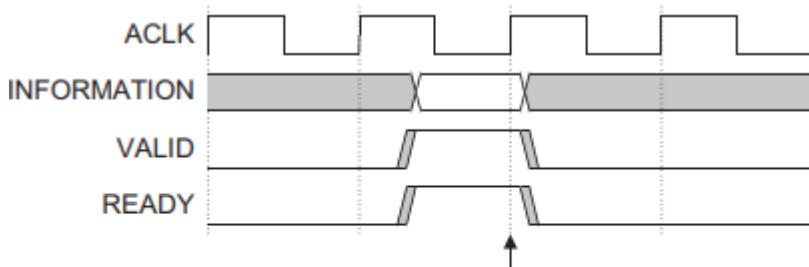
#### READY before VALID Handshake:



**Figure 3-2 READY before VALID handshake**

- The destination first indicates readiness to accept data by setting READY high.
- The source then presents data or control information, setting the VALID signal high.
- The transfer occurs when both the VALID and READY signals are high.

#### VALID with READY Handshake:



**Figure 3-3 VALID with READY handshake**

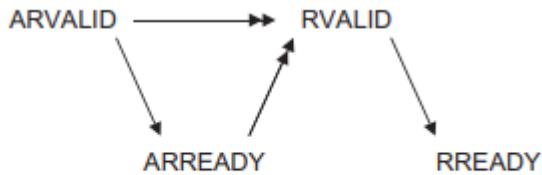
- The source and destination simultaneously indicate data availability and readiness to accept, respectively.
- Both VALID and READY signals go high in the same cycle, allowing immediate data transfer.

#### Dependencies between channel handshake signals

- **Avoiding Deadlocks:** It's crucial to observe the handshake signals' dependencies to prevent deadlocks.
- **Transaction Rules:**
  - a. VALID signals should not be dependent on READY signals of other components in the transaction.
  - b. The READY signal can wait for the VALID signal.

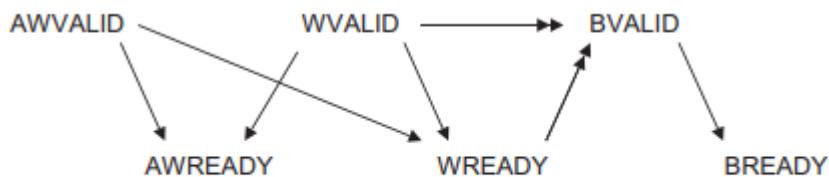
#### Read Transaction:

The starting end in the direction indicates the dependency(precondition)

**Figure 3-4 Read transaction handshake dependencies**

- Slave waits for ARVALID before asserting ARREADY.
- Slave waits for both ARVALID and ARREADY before RVALID.

Write Transaction:

**Figure 3-5 Write transaction handshake dependencies**

- Master should not wait for AWREADY or WREADY before asserting AWVALID or WVALID.
- Slave waits for AWVALID or WVALID before asserting AWREADY.

## Addressing Options

- **Responsibility:** The slave is responsible for calculating the addresses of subsequent transfers in the burst.
- **Boundary Limits:** Bursts should not cross 4KB boundaries to avoid crossing between slaves and to minimize the address incrementer within slaves.

## Burst Length

- **Signals:** AWLEN or ARLEN signals indicate the number of data transfers in a burst, ranging from 1 to 16 transfers.
- **Encoding:** The table details how binary codes (b0000 to b1111) correspond to the number of data transfers.

**Table 4-1 Burst length encoding**

<b>ARLEN[3:0]</b>	<b>Number of data transfers</b>
b0000	1
b0001	2
b0010	3
.	
.	
.	
b1101	14
b1110	15
b1111	16

- **Transaction Rules:**

- Each transaction must use the number of transfers specified by ARLEN or AWLEN without early termination.
- The master can discard further writing and read data but it must complete the remaining transfers in the burst
- **Caution:** Avoid using burst lengths that exceed the requirements of FIFO-type devices to prevent data loss.

### Burst Size

- **Signals:** ARSIZE and AWSIZE signals define the maximum number of bytes in each transfer within a burst.
- **Encoding:** The table outlines the binary codes (b0000 to b111) that correspond to bytes per transfer.

**Table 4-2 Burst size encoding**

<b>ARSIZE[2:0]</b>	<b>Bytes in transfer</b>
<b>AWSIZE[2:0]</b>	
b000	1
b001	2
b010	4
b011	8
b100	16
b101	32
b110	64
b111	128

- **Addressing:** The AXI determines which byte lanes of the data bus to use for each transfer based on the transfer address.
- **Data Transfer Consistency:** For incrementing or wrapping bursts with transfer sizes narrower than the data bus, data transfers are on different byte lanes for each beat of the burst. The address of a fixed burst remains constant, and every transfer uses the same byte lanes.
- **Transfer Size Limitations:** The size of any transfer must not exceed the data bus width of the components in the transaction.

### Burst Type

Three types of bursts for data transfer according to how the address for each transfer is handled:

1. **Fixed Burst:** The address remains constant for every transfer in the burst, suitable for repeated access to the same location, like FIFO-type access.
2. **Incrementing Burst:** The address increments from the previous one. (Works like sequential memory access)
3. **Wrapping Burst:** The address increments from the previous one until it hits a wrap boundary(similar to incrementing burst), after which it wraps around to a lower address. (Like the circular queue)

Two restrictions apply to wrapping bursts:

- The start address must be aligned to the size of the transfer.
- The burst length must be a power of two, specifically 2, 4, 8, or 16.

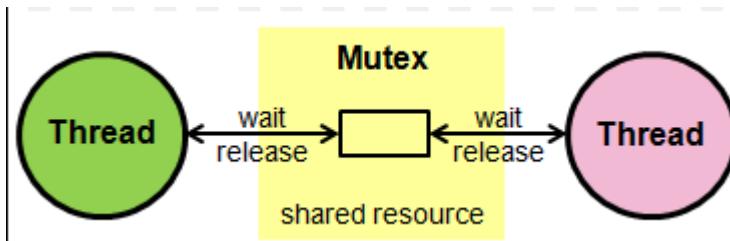
## Atomic Accesses

**Table 6-1 Atomic access encoding**

ARLOCK[1:0]	Access type
AWLOCK[1:0]	
b00	Normal access
b01	Exclusive access
b10	Locked access
b11	Reserved

The ARLOCK[1:0] or AWLOCK[1:0] signal provides exclusive access and locked access.

### Exclusive Access



**Exclusive access** notifies if a specific location has been altered by another master between a read and write operation by the initiating master. This semaphore type operation does not critically impact either access latency or the bandwidth

- Responses BRESP[1:0] or RRESP[1:0] indicate the success or failure of exclusive operations.
- A fail-safe mechanism is provided to indicate when a master attempts an exclusive access to a slave that does not support it.

### Exclusive Access Process

1. A master performs an exclusive read from an address location.
2. At some later time, the master attempts to complete the exclusive operation by performing an exclusive write to the same address location.
3. The exclusive write access of the master is signalled as:
  - **Successful** if no other master has written to that location between the read and write accesses.

- **Failed** if another master has written to that location between the read and write accesses. In this case the address location is not updated.

## Locked Access

**Locked access** ensures a group of transactions occurs without any other access by other masters in between, where the bus interconnect locks the bus to the master that initiated the locked transaction.

### Key Points:

#### 1. Signal for Locked Transfer:

- When ARLOCK[1:0] or AWLOCK[1:0] signals indicate a locked transfer, it is the interconnect's responsibility to ensure exclusive access to the initiating master until an unlocked transfer completes.

#### 2. Starting a Locked Sequence:

- A master initiating a locked sequence must not have other outstanding transactions pending.

#### 3. Completing a Locked Sequence:

- All previous locked transactions must be completed before issuing the final unlocking transaction. The final unlocking transaction is necessary before any further transactions begin.

### Notes:

- Locked accesses can affect interconnect performance as they prevent other transactions during the sequence.
- It is advised to use locked accesses only for legacy devices due to their impact.

## Response Signaling

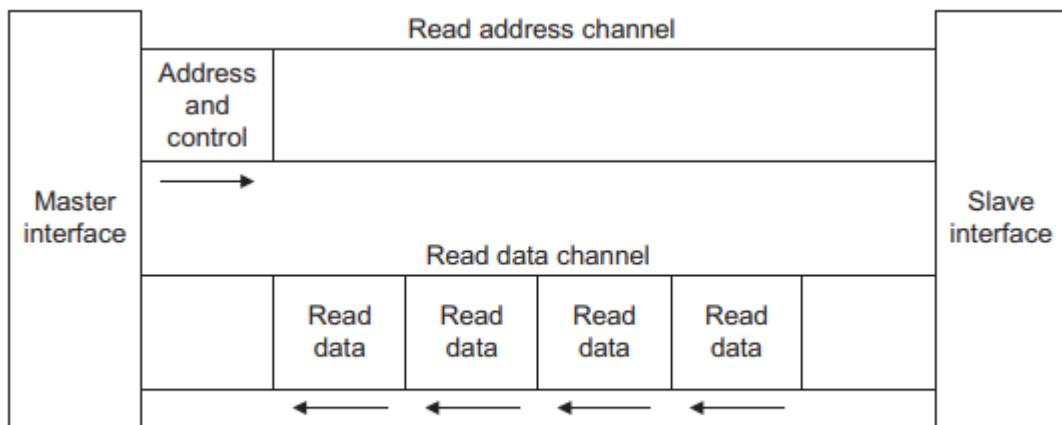
The AXI protocol supports response signaling for both read and write operations.

### Responses Defined:

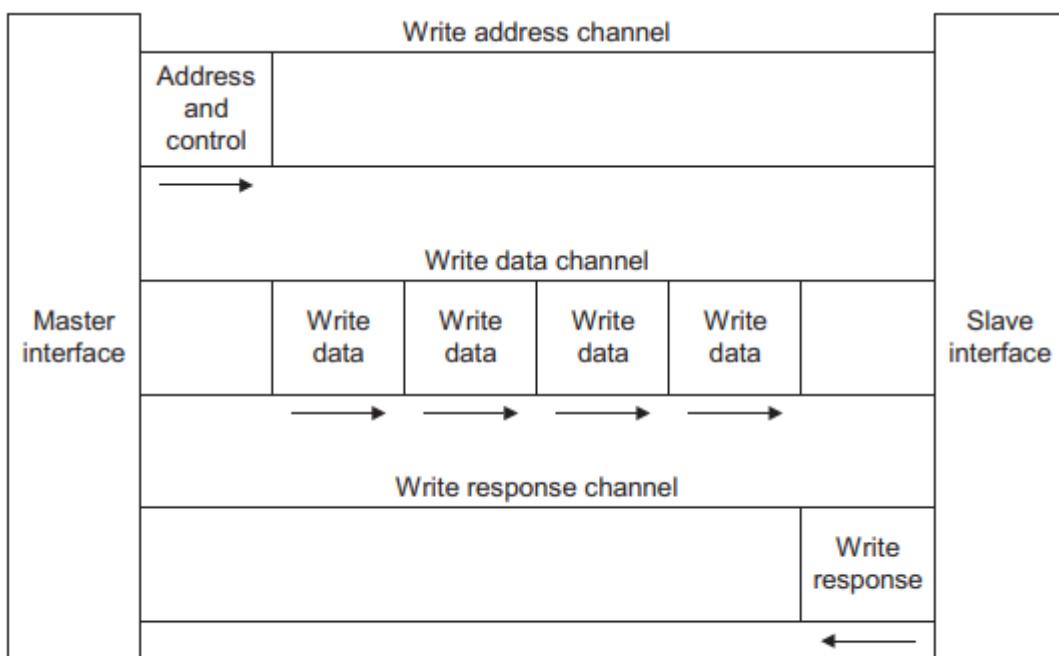
- **OKAY(b00)**: Indicates a normal access has been successful. It can also signify the absence of an exclusive access failure.
- **EXOKAY(b01)**: Marks that an exclusive access read or write operation was successful.
- **SLVERR(b10)**: Signifies a successful access to the slave, but the slave cannot perform the requested action and is returning an error.
- **DECERR(b11)**: Indicates a decode error, which occurs typically when no slave is present at the specified transaction address.

### Response channel:

- **When data is read**, the response from the slave (the peripheral or memory being accessed) is sent along with the data itself.

**Figure 1-1 Channel architecture of reads**

- For write operations, the response is sent on a separate write response channel.

**Figure 1-2 Channel architecture of writes**

### Number of responses

- For write operations, there's only one response for the entire burst of data transfers, not for each transfer within the burst.
- For read transactions, the slave can signal different responses for each data transfer within a burst.

## Ordering Model

### About the Ordering Model (Similar to pipeline out of order)

The AXI protocol allows for out-of-order completion and issuing of multiple outstanding addresses. This flexibility optimizes data throughput and system efficiency by allowing:

- **Out-of-Order Transaction Completion:** Enables transactions to faster memory regions to complete without having to wait for transactions to slower memory regions, thus reducing the effect of transaction latency.
- **Multiple Outstanding Addresses:** Masters can issue transaction addresses without waiting for earlier transactions to complete, allowing for parallel processing and transactions.

## Transfer ID Fields

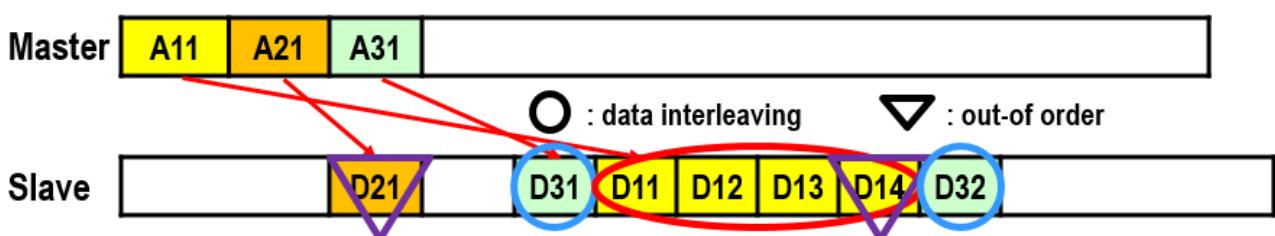
The AXI protocol uses ID fields to manage multiple transactions and maintain order:

- **ARID/AWID Field:** Specifies additional ordering requirements from the master.

### Ordering Rules:

- Transactions from different masters have no ordering restrictions.
- Transactions with different IDs from the same master can complete in any order.
- Write transactions with the same AWID must complete in the same order as the master issued the addresses.
- Like the write transactions, read transactions with the same ARID must return data in the same order as addresses were issued.

## Data Interleaving



Write data interleaving allows a slave to handle interleaved write data with different AWID values. This facilitates the simultaneous processing of write data from multiple sources, which can enhance system performance by allowing data from slower sources to be interspersed with data from faster ones.

- **Write Data Interleaving Depth:** This is a static value set by the designer that indicates how many different addresses can have their write data interleaved in the slave interface. A depth of 1 means no interleaving is possible, while a higher value allows for multiple interleaved transactions.
- **AWID as a single unit:** Interleaving is not allowed between transactions with the same AWID. However, data with different AWID values can be interleaved.

- **Avoiding Deadlocks:** To prevent deadlocks, a slave interface with an interleaving depth greater than one must be able to continuously accept interleaved data without stalling.

## Data buses

### About the data buses

- Each data bus has its own set of handshake signals, which allows for simultaneous data transfers on both buses.
- **Data width not over bus width:** Every transfer generated by a master must be the same width as or narrower than the data bus for the transfer.

### Write strobes

63	56 55	48 47	40 39	32 31	24 23	16 15	8 7	0
7	6	5	4	3	2	1	0	

Figure 9-1 Byte lane mapping

Each write strobe signal, `WSTRB`, corresponds to one byte of the write data bus.

When asserted, a write strobe indicates that the corresponding byte lane of the data bus contains valid information to be updated in memory, which facilitates selective data transfer.

### Narrow Transfers

When a master generates a **transfer that is narrower than its data bus**, the address and control information determine which byte lanes the transfer uses.

- **In incrementing or wrapping burst modes**, different byte lanes may be used for each beat of the burst.
- **In a fixed burst**, the address remains constant, and the byte lanes that can be used also remain constant.

### Example of Byte Lanes Use

In Figure 9-2:

- the burst has five transfers
- the starting address is 0
- each transfer is eight bits
- the transfers are on a 32-bit bus.

Byte lane used			
			DATA[7:0]
		DATA[15:8]	
	DATA[23:16]		
DATA[31:24]			
			DATA[7:0]

1st transfer  
2nd transfer  
3rd transfer  
4th transfer  
5th transfer

Figure 9-2 Narrow transfer example with 8-bit transfers

## Byte Invariance

Byte invariance refers to a scheme that allows for access to mixed-endian data structures within the same memory space.

Byte-invariant endianness means that a byte transfer to a given address passes the 8 bits of data on the same data bus wires to the same address location.

### Endian Compatibility:

- Little-endian components can usually connect directly to a byte-invariant interface.
- Big-endian components require a conversion function for byte-invariant operations.

### Example:

Figure 9-4 illustrates a data structure requiring byte-invariant access. Header information like source and destination identifiers may be in little-endian, whereas the payload is in big-endian format.

31	24 23	8 7	0
Desti-	Source	Packet	
Checksum			-nation
Payload		Data items	
Payload			
Payload			
Payload			

Figure 9-4 Example mixed-endian data structure

### Importance of Byte Invariance:

Ensures that little-endian access to parts of the header does not corrupt the big-endian data within the structure.

## Unaligned Transfer

### About unaligned transfers

#### Alignment

- **Typically aligned:** Typically, each data transfer is aligned to the size of the transfer. For example, a 32-bit wide transfer is usually aligned to four-byte boundaries. However, there are times when it is desirable to begin a burst at an unaligned address.
- **Sometimes unaligned:** For any burst that is made up of data transfers wider than one byte, it is possible that the first bytes accessed do not align with the natural data width boundary. For example, a 32-bit (four-byte) data packet that starts at a byte address of 0x1002 is not aligned to a 32-bit boundary.

#### Note on Slave Actions:

- The protocol does not require the slave to take special actions based on alignment information provided by the master.

#### Example

These are examples of aligned and unaligned transfers on buses with different widths.

- Each row in the figures represents a transfer.
- The shaded cells indicate bytes that are not transferred, based on the address and control information.

	31	24	23	16	15	8	7	0	
Address: 0x00	3	2	1	0					1st transfer
Transfer size: 32 bits	7	6	5	4					2nd transfer
Burst type: incrementing	B	A	9	8					3rd transfer
Burst length: 4 transfers	F	E	D	C					4th transfer
Address: 0x01	3	2	1	0					1st transfer
Transfer size: 32 bits	7	6	5	4					2nd transfer
Burst type: incrementing	B	A	9	8					3rd transfer
Burst length: 4 transfers	F	E	D	C					4th transfer
Address: 0x01	3	2	1	0					1st transfer
Transfer size: 32 bits	7	6	5	4					2nd transfer
Burst type: incrementing	B	A	9	8					3rd transfer
Burst length: 5 transfers	F	E	D	C					4th transfer
	13	12	11	10					5th transfer
Address: 0x07	7	6	5	4					1st transfer
Transfer size: 32 bits	B	A	9	8					2nd transfer
Burst type: incrementing	F	E	D	C					3rd transfer
Burst length: 5 transfers	13	12	11	10					4th transfer
	17	16	15	14					5th transfer

**Figure 10-1 Aligned and unaligned word transfers on a 32-bit bus**

- First burst: **aligned** (Transfer size: 4 bytes, Address: 0x00)
- Second burst: **unaligned** (Transfer size: 4 bytes, Address: 0x01)

## Clock and Reset

### Clock and Reset

- Requirements for implementing the ACLKn and ARESETN signals.

## Clock

- Each AXI component uses a single clock signal, ACLKn.
- All input signals are sampled on the rising edge of ACLKn.

- All output signal changes must occur after the rising edge of ACLKn.

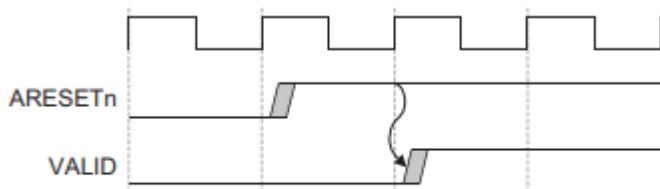
## Reset

- AXI protocol includes a single **active LOW** reset signal, ARESETn.
- ARESETn can be asserted asynchronously, but deassertion must be synchronous after the rising edge of ACLKn.
- Interface requirements during reset:
  - A master interface must drive ARVALID, AWVALID, and WVALID *LOW*.
  - A slave interface must drive RVALID and BVALID *LOW*.

## Exit from reset

A master interface must begin driving ARVALID, AWVALID, or WVALID HIGH only at a rising ACLK edge after ARESETn is HIGH

Figure 11-1 shows the first point after reset that VALID can be driven HIGH.



**Figure 11-1 Exit from reset**

## Low-power Interface

### About the Low-Power Interface

The low-power interface is an optional extension to the data transfer protocol that targets two different classes of peripherals:

- **Peripherals Requiring Power-Down Sequence:**
  - These peripherals need to enter a low-power state before their clocks can be turned off.
  - They require a signal from the system clock controller to start the power-down sequence.
- **Peripherals Without Power-Down Sequence:**
  - These can independently indicate when their clocks can be turned off without a sequence.

### Low-power clock control

#### Overview:

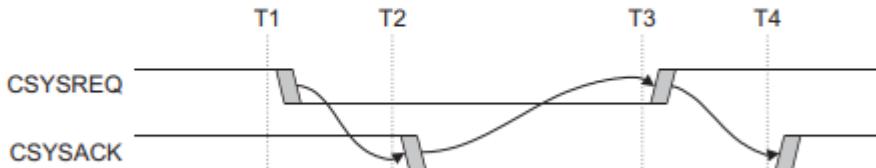
- The low-power clock control interface consists of signals from the peripheral and the system clock controller for managing the clock in low-power states.

#### Signals:

- CACTIVE:** Signal from the peripheral indicating that it requires its clock to be either enabled(HIGH) or disabled(LOW).
- CSYSREQ(request):** Handshake signal for the system clock controller to request entry(LOW) into or exit(HIGH) from a low-power state.
- CSYSACK(acknowledge):** Handshake signal(LOW) for the peripheral to acknowledge the low-power state request or the exit from it.

#### Example

Figure 12-1 shows the relationship between **CSYSREQ** and **CSYSACK**.



**Figure 12-1 CSYSREQ and CSYSACK handshake**

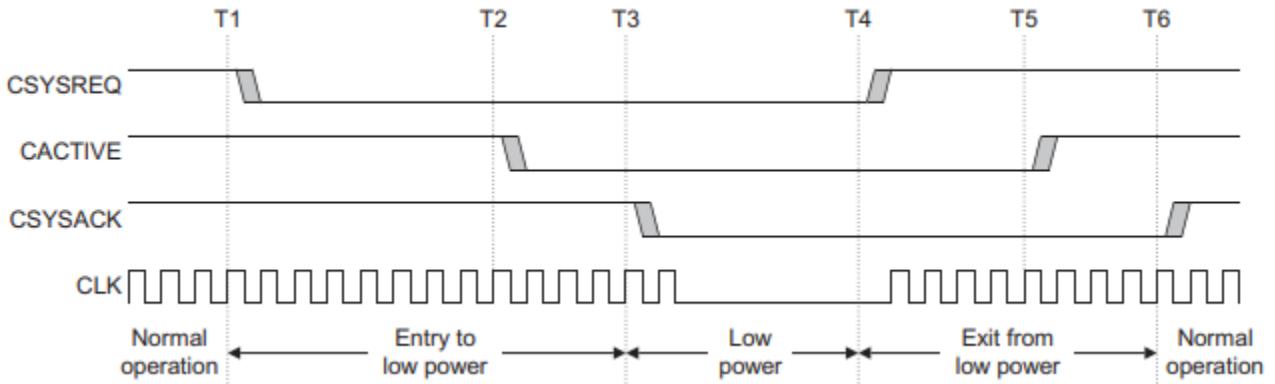
- T1:** Request to put the peripheral in a low-power state
- T2:** Request acknowledged
- T3:** Exit from the low-power state
- T4:** Exit acknowledged

#### Acceptance of low-power request

#### Handshake Mechanism:

- The sequence of events when a peripheral accepts a system low-power request is outlined in Figure 12-2.

#### Example:

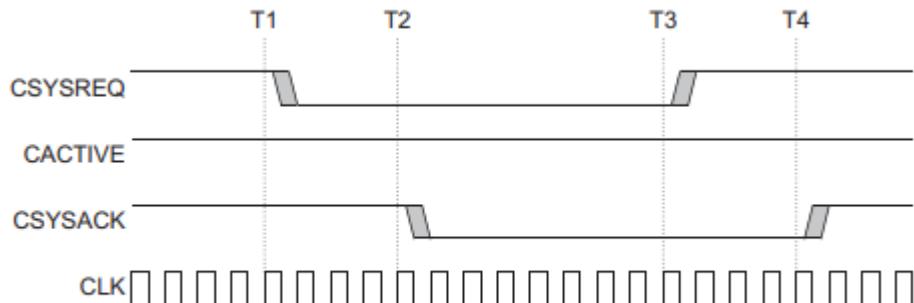


- **T1:** Request to put the peripheral in a low-power state
- **T2-T3:** The peripheral deasserts CACTIVE as it performs its power-down function and acknowledges by deasserting CSYSACK
- **T4:** Exit from the low-power state
- **T5-T6:** The peripheral asserts CACTIVE and completes the exit sequence by asserting CSYSACK.

#### Denial of a low-power request

- A low-power request is denied by the peripheral by keeping CACTIVE HIGH when acknowledging the request with CSYSACK.

#### Example:



**Figure 12-3 Denial of a low-power request**

- **T1:** Request to enter a low-power state.
- **T2:** Peripheral denies the request by keeping CACTIVE HIGH when it acknowledges the request.
- **T3-T4:** CSYSREQ is asserted to complete the sequence. The handshake must be completed by asserting CSYSREQ before another request can be initiated.

#### 12.2.4 Clock control sequence summary

Figure 12-4 shows the typical flow for entering and exiting a low-power state.

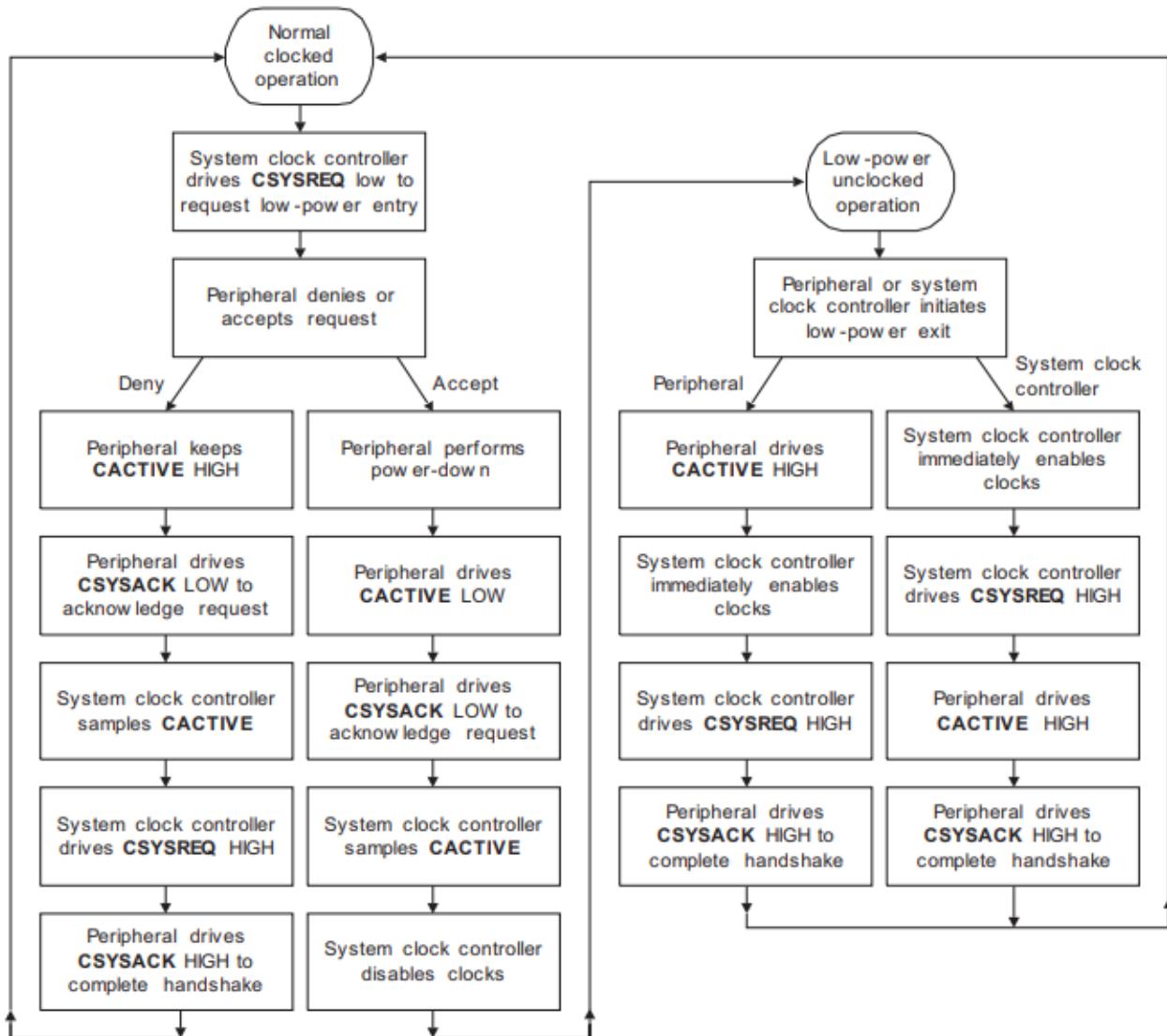


Figure 12-4 Low-power clock control sequence

# Linux

## 12.19 과제

### Chapter 1

7번: the kernel, the shell, and the programs.

8번: 가상머신은 커널을 포함해 OS를 완전히 시뮬레이팅하며, docker와같은 컨테이너와 달리 overhead가 크다.

### Chapter 2

1: 3

2: 2

3: 4

4: 2 (같은 inode를 가리키는 hard link가 2개이므로)

5: 3 (~: 홈 디렉토리, -name f1: f1 찾기, -ok rm {} \: 삭제) => 보기ガ 다 틀렸음. 정답은 **find ~ -name f1 -exec rm -i {} \;**

6: 2

7: 3

8: /, ..

9: Regular, directory, device, link

10: 해당 사용자에게 할당된 개인적인 파일 저장 공간, 사용자 계정 생성 시, 시스템 관리자에 의해 수동 설정

11:

명령	의미
cd .../temp	사용자의 홈 디렉토리로 이동 → 상위디렉토리 아래 있는 temp 디렉토리로 이동
cd ~user2	루트 디렉토리에 있는 temp로 이동 => user2의 홈 디렉토리로 이동
cd ~/temp	user2사용자의 홈디렉토리로 이동 => 홈디렉토리 아래 temp 디렉토리로 이동
cd /tmp	루트 디렉토리의 tmp로 이동

12: -F

13: symbolic link

14: 비밀파일, ls -a로 모든 파일 출력

15: -l => **ls -d / 또는 ls -dl /**

16: 파일 시스템의 최상위부터 시작하는 전체 경로, 현재 작업 중인 디렉토리를 기준으로 하는 경로

17: 하드링크는 원본 파일과 동일한 데이터(inode)를 가리킴, 심볼릭링크는 원본 파일의 경로를 저장, 복사파일은 원본 파일의 데이터를 새로운 위치에 복사하여 새로운 파일을 생성

18: rmdir는 비어 있는 디렉토리만 삭제할 수 있다, rm -r은 recursive의 약자로 디렉토리와 그 내부의 모든 파일 및 하위 디렉토리를 재귀적으로 삭제한다.

19: 모든 directory는 기본적으로 두 개의 하드 링크를 가진다(자기 자신(.)과 상위 directory(..)) 그러므로 temp의 하드 링크는 2개이고 ..의 하드링크는 temp를 포함해 3개이다.

20: 심볼릭 링크는 원본 파일의 경로만 저장하기 때문에, 원본 파일이 삭제되었다 해도 링크 자체는 그대로 남아있다. 이후에 새로운 파일을 원본 파일의 이름으로 복사하면, 심볼릭 링크는 이 새로운 파일을 가리킨다. 그러므로 C파일의 내용이 출력된다.

21: source파일 3개를 temp 디렉토리로 복사하는 것을 의미한다. temp는 directory어야 한다.

22: inode는 파일의 중요한 정보를 저장한다. Metadata, 실제 데이터의 주소를 저장한다.

23: hard link임을 의미한다. 한 파일의 hard link들은 같은 inode를 가리킨다.

24:

파일	절대 경로명	상대 경로명
/	/	./
lib	/usr/lib	../../../../../usr/lib
data1	/home/user1/ch3/data1	..//ch3/data1
test	/home/user1/ch2/test	./test
hosts	/etc/hosts	../../../../etc/hosts

25. /usr/lib (절대경로)

26. ch2는 비어있지 않기때문에 rm -r을 이용해 재귀적으로 삭제해야한다.

**27번 부터 안풀었네!!!**

## 12.20 과제



practice\_questions.txt



screen.txt

## 12.21,22 과제

### 1번 문제

```
#!/bin/bash

# Extract the line containing 'Kernel command line' from dmesg
kernel_cmd_line=$(dmesg | grep "Kernel command line:")

# Remove everything up to and including ':' and space
cut_line=${kernel_cmd_line#: }

# Split the cut_line on spaces, then iterate and extract values after '='
for piece in $cut_line; do # includes .split()
    if [[ $piece == *=* ]]; then
        # Cut the string up to '=', and print the remaining part
        echo ${piece#=}
    fi
done
```

### 1번 문제 python

```
# Python code to achieve similar functionality
import subprocess
import re

def extract_values_from_kernel_cmd_line():
    # Run the dmesg command and capture its output
    dmesg_output = subprocess.check_output("sudo dmesg", text=True, shell=True)

    # Capture the part behind 'Kernel command line'
    kernel_cmd_line = re.search(r'Kernel command line: (.*)', dmesg_output)
    if kernel_cmd_line:
        cmd_line = kernel_cmd_line.group(1)

        # Extract values behind '=' and return them
        return re.findall(r'=([^\s]+)', cmd_line)
    else:
        return []

# Call the function and print the results
values = extract_values_from_kernel_cmd_line()
for value in values:
    print(value)
```

### 3번 문제

```
#!/bin/bash

# Infinite loop to print the current time every 2 seconds
while true; do
    echo "Current time: $(date)"
    sleep 2
done
```

## 12.26 chapter 06

1. ps -f
2. **ps -u han1**: This command lists the processes for the user han1. **ps -fu han1**: This command provides a full-format listing of processes for the user han1, including more detailed information. **ps -p \$(pgrep -u han1)**: This command uses pgrep -u han1 to find process IDs of all processes belonging to user han1 and then uses ps -p to show details of these processes. **ps -pu han1**
3. **fg %2**: This command correctly references the second job using %2, which is the standard way to refer to a job number in Unix-like systems. **fg**: Without any arguments, this would bring the most recent job to the foreground, which is not necessarily job number 2. **fg \$2**: This syntax is incorrect for job control. \$2 would typically be used to reference the second argument in a shell script or function,

not a job number. **fg #2:** This syntax is also incorrect. The # symbol is generally used for comments in shell scripts and would not be used to reference a job number.

4. 2
5. 4 UID는 user의 ID이다.
6. 2

8: Daemon processes run in the background, often starting with the system, and do not interact directly with users. In contrast, regular processes are usually initiated by users and run in the foreground.

9: 61->21->10->01

10: ps -ef | grep '^guest'

11: kill -9 5000 (signal)

12:

#### Shell script

```
while true; do
    ps -ef
    sleep 2
done
```

14: 1. Suspend the foreground process by pressing Ctrl+Z. 2. Resume the process in the background by entering 'bg'.

15: Identify the background job by using the 'jobs' command, which lists all jobs with their respective job numbers. 2. Bring the job to the foreground by using the 'fg' command followed by the job number.

16: kill %3

17: nohup find / -name test &

19: since guest01 is listed in /etc/cron.allow, they are allowed to use crontab, irrespective of their presence in /etc/cron.deny. The cron.allow file takes precedence over cron.deny in granting permissions.

20: 1. at 12:00 PM Dec 31 2. Type ps -u user01 3. Press Ctrl+D

## 12.27 과제

2

ip route add default via 192.168.10.1 dev ens33

3

```
hostnamectl set-hostname web.server
```

4

```
arp 192.168.10.1
```

5

192.168.100.0

7

1

The `-h` option in the `uname` command displays help information about the command itself, not for retrieving the hostname.

8

A protocol is a type of 'promise' consisting of a set of rules or standards that define how data is transmitted and received across a network. Examples of well-known protocols include HTTP (Hypertext Transfer Protocol) for web traffic, SMTP (Simple Mail Transfer Protocol) for email, and TCP/IP (Transmission Control Protocol/Internet Protocol) for general internet data transmission. Protocols are needed to standardize how different systems can communicate effectively.

9

while both MAC and IP addresses are used to identify devices on a network, they operate at different layers of the network, have different formats and purposes, and are assigned in different ways. MAC addresses are more about the physical hardware identification, and IP addresses are more about the logical network location and routing.

10

The primary function of a netmask is to divide an IP address into the network and host portions. This is crucial for understanding which part of an IP address identifies the network and which part identifies a specific device (host) within that network.

**11**

Changing the subnet mask of a Class C IP address from `255.255.255.0` to `255.255.255.128` divides the network into two smaller subnets, each with up to 128 addresses.

**18**

```
ip route add default via 10.10.10.254
```

- `ip` : This is the command used for manipulating routing, devices, policy routing, and tunnels.
- `route add` : This part of the command adds a new route to the kernel routing tables.
- `default` : Specifies that this route is the default route, meaning it will be used when no other route matches.
- `via 10.10.10.254` : This specifies that the gateway for this default route is `10.10.10.254` .)

**19**

```
ifconfig eth1 10.10.10.5 netmask 255.255.255.128 up
```

- `ifconfig` : This is the command used for configuring network interfaces.
- `eth1` : This is the name of the network interface you want to configure.
- `10.10.10.5` : This is the IP address you are assigning to the interface `eth1` .
- `netmask 255.255.255.128` : This sets the subnet mask for the interface. In this case, the subnet mask is `255.255.255.128` , which is a standard notation.
- `up` : This option activates the network interface after configuring it.

**27**

```
netstat -s
```

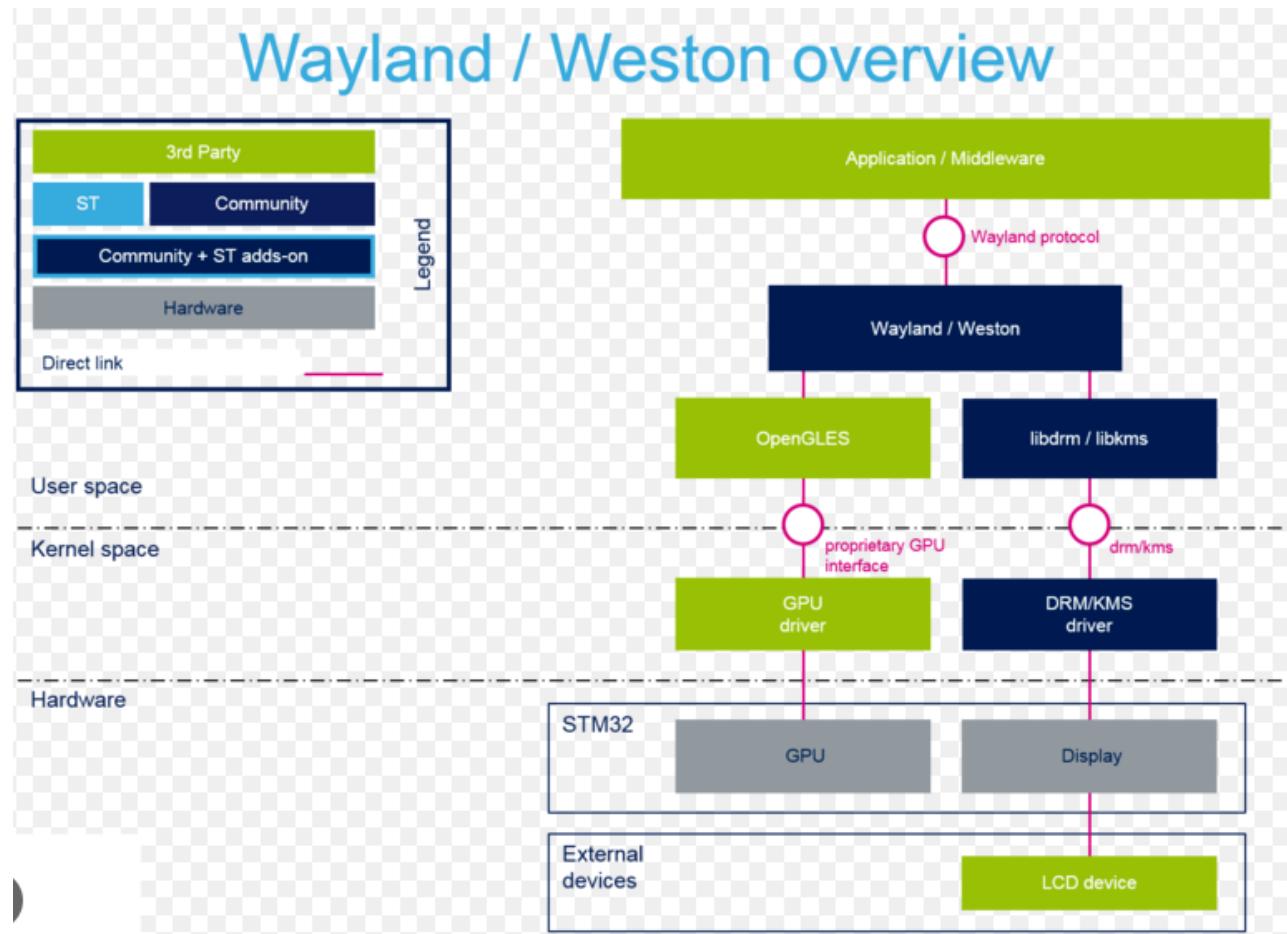
This command provides a detailed summary of all the statistics for each protocol supported by `netstat`.

# Yocto 양영식

## 1st exercise

1: Create and execute the image of Arm64 Weston/wayland

**What is Weston/wayland?**



- Wayland is a **protocol** that defines the communication between a display server (compositor) and its clients (applications). It's the modern replacement for the older X11 protocol used by the X Window System.
- Weston is the reference **implementation** of a Wayland compositor. In the Wayland architecture, the compositor is a key component that manages the display, controls window positioning, and handles input events, among other tasks.

## Environment setup

Install dependencies using apt-get install

```
sudo apt-get update
sudo apt-get install gawk wget git-core diffstat unzip texinfo gcc-multilib build-essential chrpath socat cpio python3 python3-pip python3-pexpect xz-utils debianutils iputils-ping python3-git python3-jinja2 libegl1-mesa libSDL1.2-dev xterm python3-subunit mesa-common-dev lz4
```

## Source download

Download poky (kirkstone) because dunfell results in some issues

```
git clone -b kirkstone git://git.yoctoproject.org/poky.git
```

## Modify

Modify this line in `mybuild/conf/local.conf` so that ARM64 is used

```
MACHINE ?= "qemuarm64"
```

Optional (It works well without these line because of the default setting)

```
DISTRO_FEATURES:append = " wayland"
CORE_IMAGE_EXTRA_INSTALL += "wayland weston"
```

## How to build

Build using Poky and Bitbake.

```
source poky/oe-init-build-env ← Sets the build environment
```

```
source poky/oe-init-build-env mybuild
cd mybuild
bitbake -k core-image-weston
```

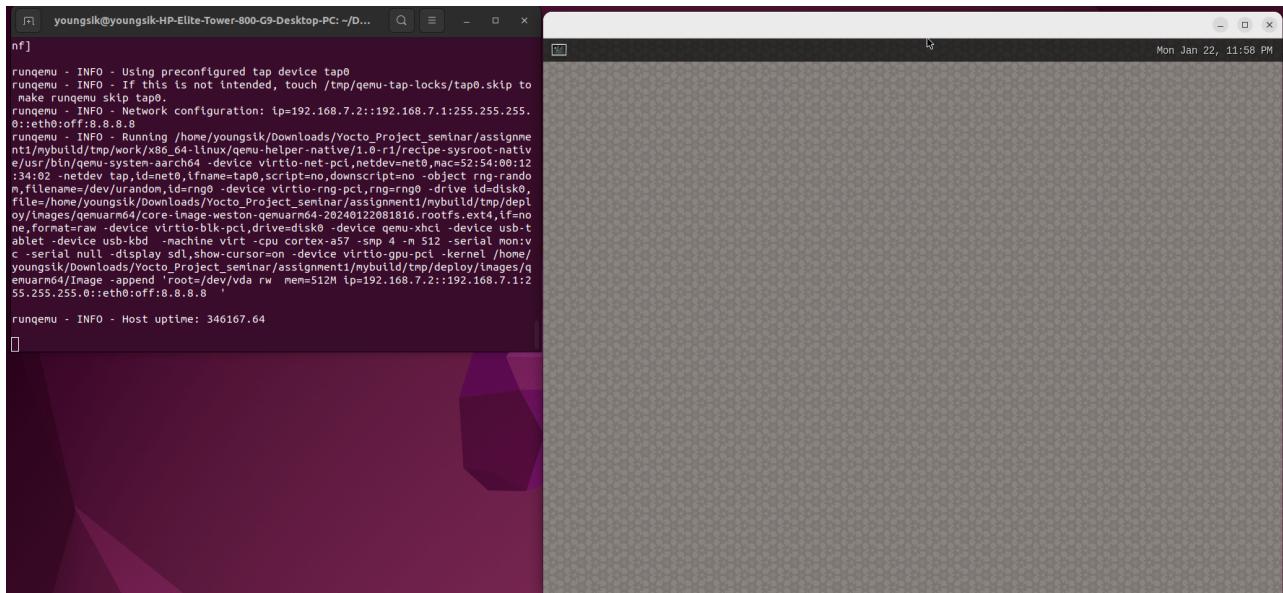
## How to run

Run the built image using QEMU

```
runqemu qemuarm64
```

QEMU is a free and open-source emulator. It emulates a computer's processor through dynamic binary translation and provides a set of different hardware

## Results



## Trouble shoot

- Got an error saying it cannot continue the build without lz4, and solved it by installing lz4
- Got an error because of using DISTRO\_FEATURES as described the official document of Dunfell and solved it by using **DISTRO\_FEATURES:append = " wayland"** from the kirkstone document

2:

**source 와 ./ 의 차이는?**

- `Source` is used to execute the environment setup script (e.g. `source oe-init-build-env`)
- Using `./` does not achieve this

## Embedded Linux GUI 의 종류와 각각의 설명 (요약)

### 1. Qt for Embedded Linux

- **Description:** Qt for Embedded Linux is a popular framework that offers a comprehensive set of tools and libraries for developing GUIs. It's known for its rich set of features and cross-platform support.

### 2. GTK+ (GIMP Toolkit)

- **Description:** GTK+ is a free and open-source cross-platform widget toolkit for creating graphical user interfaces. It's part of the GNU Project and originally developed for the GIMP image editor.

### 3. X Window System (X11)

- **Description:** X11 is a windowing system for bitmap displays. It is the standard window system for UNIX and UNIX-like operating systems, including Linux.

### 4. Wayland

- **Description:** Wayland is a modern protocol for a display server, intended as a simpler replacement for X11. It's gaining popularity in the Linux world for its efficiency.

### 5. FLTK (Fast Light Toolkit)

- **Description:** FLTK is a lightweight, cross-platform GUI toolkit. It's known for its small size and speed, making it suitable for embedded systems with limited resources.

### 6. Embedded Wizard

- **Description:** Embedded Wizard is a tool for developing high-performance GUIs specifically for embedded systems.

## 2nd exercise

1:

### Set up the environment

Download base poky

```
git clone -b kirkstone git://git.yoctoproject.org/poky.git
```

Set the build environment

```
source poky/oe-init-build-env ; cd ..
```

### Create a custom layer

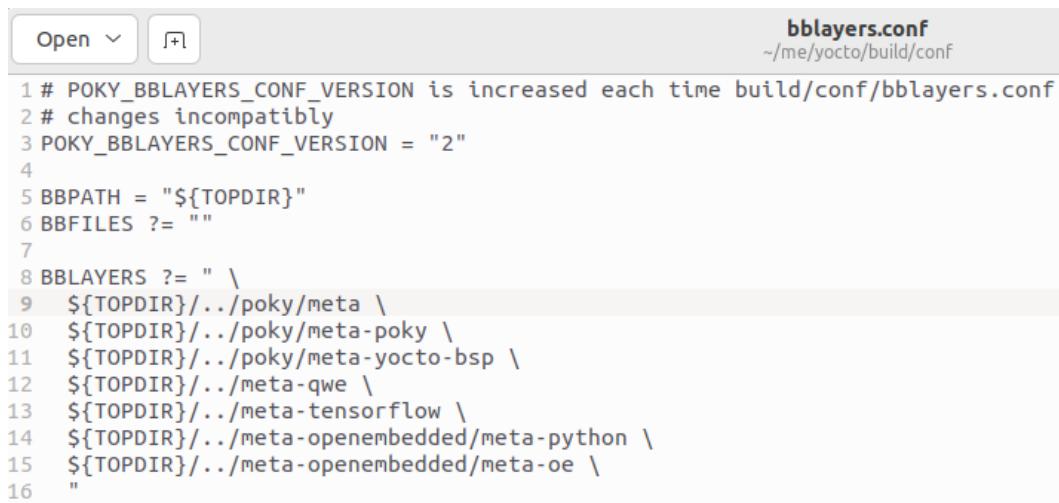
Create a layer

meta-[name] (e.g. meta-qwe)

```
bitbake-layers create-layer meta-[name]
```

Add the layer

Add dependency layers in `build/conf/bblayers.conf`



```

Open ▾ bblayers.conf  

~/me/yocto/build/conf
1 # POKY_BBLAYERS_CONF_VERSION is increased each time build/conf/bblayers.conf
2 # changes incompatibly
3 POKY_BBLAYERS_CONF_VERSION = "2"
4
5 BBPATH = "${TOPDIR}"
6 BBFILES ?= ""
7
8 BBLAYERS ?= " \
9 ${TOPDIR}../poky/meta \
10 ${TOPDIR}../poky/meta-poky \
11 ${TOPDIR}../poky/meta-yocto-bsp \
12 ${TOPDIR}../meta-qwe \
13 ${TOPDIR}../meta-tensorflow \
14 ${TOPDIR}../meta-openembedded/meta-python \
15 ${TOPDIR}../meta-openembedded/meta-oe \
16 "

```

## Create a core image recipe

Create an "images" directory in the layer created above

```
mkdir -p [root]/meta-[name]/recipes-core/images
```

Create [name]-image.bb in this directory and write the image recipe in it

```

DESCRIPTION = "A core image for QWE"
LICENSE = "MIT"

# Core files for basic console boot
IMAGE_INSTALL = "packagegroup-core-boot"

# Add our desired extra files
IMAGE_INSTALL += "psplash dropbear"

inherit core-image

IMAGE_ROOTFS_SIZE ?= "8192"

```

## Create a hello world package

Create hello world source

```
mkdir -p [root]/meta-[name]/recipes-core/hello/files
touch [root]/meta-[name]/recipes-core/hello/files/hello.c
```

### hello.c

```
#include <stdio.h>

int main(int argc, char **argv) {
    printf("Hello World\n");
    return 0;
}
```

Create a hello world recipe

```
touch [root]/meta-[name]/recipes-core/hello/hello_1.0.bb
```

```
DESCRIPTION = "Hello World example"
LICENSE = "MIT"

LIC_FILES_CHKSUM = "file://${COREBASE}/meta/
COPYING.MIT;md5=3da9cfbcb788c80a0384361b4de20420"

S = "${WORKDIR}"

SRC_URI = "file://hello.c"

do_compile() {
    ${CC} ${CFLAGS} ${LDFLAGS} hello.c -o hello
}

do_install() {
    install -d -m 0755 ${D}/${bindir}
    install -m 0755 hello ${D}/${bindir}/hello
}
```

## Add the hello world package to the core image recipe

```
[root]/meta-[name]/recipes-core/images/[name]-image.bb
```



```
qwe-image.bb
~/Downloads/me/Yocto_Project_seminar/assignment1/meta-qwe/recipes-core/images
Open Save
1 DESCRIPTION = "A core image for QWE"
2 LICENSE = "MIT"
3
4 # Core files for basic console boot
5 IMAGE_INSTALL = "packagegroup-core-boot"
6
7 # Add our desired extra files
8 IMAGE_INSTALL += "psplash dropbear hello"
9
10 inherit core-image
11
12 IMAGE_ROOTFS_SIZE ?= "8192"
```

## Build and run the custom image

```
bitbake -k qwe-image
runqemu qwe-image
```

```

[ 1.319190] ata1: SATA link down (SStatus 0 SControl 300)
[ 1.321032] ata3: SATA link up 1.5 Gbps (SStatus 113 SControl 300)
[ 1.323536] ata3.00: ATAPI: QEMU DVD-ROM, 2.5+, max UDMA/100
[ 1.324760] ata3.00: applying bridge limits
[ 1.326629] ata3.00: configured for UDMA/100
[ 1.331712] scsi 2:0:0:0: CD-ROM           QEMU     QEMU DVD-ROM    2.5+ PQ: 0 ANSI: 5
[ 1.357466] sr 2:0:0:0: [sr0] scsi3-mmc drive: 4x/4x cd/rw xa/form2 tray
[ 1.358805] cdrom: Uniform CD-ROM driver Revision: 3.20
[ 1.450961] input: QEMU QEMU USB Tablet as /devices/pci0000:00/0000:00:1d.7/usb1/1-1/1-1:1.0/0003:0627:0001.0001/input/input4
[ 1.458719] hid-generic 0003:0627:0001.0001: input: USB HID v0.01 Mouse [QEMU QEMU USB Tablet] on usb-0000:00:1d.7-1/input0
[ 1.536654] input: ImExPS/2 Generic Explorer Mouse as /devices/platform/i8042/serio1/input/input3
[ 1.562346] IP-Config: Complete:
[ 1.567210]   device=eth0, hwaddr=52:54:00:12:34:02, ipaddr=192.168.7.2, mask=255.255.255.0, gw=192.168.7.1
[ 1.568938]   host=192.168.7.2, domain=, nis-domain=(none)
[ 1.570776]   bootserver=255.255.255.255, rootserver=255.255.255.255, rootpath=
[ 1.570805]   nameserver0=8.8.8.8
[ 1.578422] md: Waiting for all devices to be available before autodetect
[ 1.579673] md: If you don't use raid, use raid=nodetect
[ 1.580991] md: Autodetecting RAID arrays.
[ 1.583334] md: autorun ...
[ 1.584559] md: ... autorun DONE.
[ 1.610280] EXT4-fs (vda): recovery complete
[ 1.618208] EXT4-fs (vda): mounted filesystem with ordered data mode. Opts: (null). Quota mode: disabled.
[ 1.623419] VFS: Mounted root (ext4 filesystem) on device 253:0.
[ 1.626043] devtmpfs: mounted
[ 1.711304] tsc: Refined TSC clocksource calibration: 1996.756 MHz
[ 1.712824] clocksource: tsc: mask: 0xffffffffffff max_cycles: 0x39906c43571, max_idle_ns: 881590820589 ns
[ 1.714376] clocksource: Switched to clocksource tsc
[ 1.774445] Freeing unused kernel image (initmem) memory: 1880K
[ 1.779824] Write protecting the kernel read-only data: 22528k
[ 1.787203] Freeing unused kernel image (text/rodata gap) memory: 2036K
[ 1.793601] Freeing unused kernel image (rodata/data gap) memory: 484K
[ 1.795978] Run /sbin/init as init process
INIT: version 3.01 booting
[ 2.031367] IPv6: ADDRCONF(NETDEV_CHANGE): eth0: link becomes ready
warning: FBIOPUT_VSCREENINFO failed, double buffering disabled
Starting udev
INIT: Entering runlevel: 5
Configuring network interfaces... ip: RTNETLINK answers: File exists
Starting Dropbear SSH server: dropbear.
Starting syslogd/klogd: done

Poky (Yocto Project Reference Distro) 4.0.15 qemux86-64 /dev/tty1

qemux86-64 login: root
root@qemux86-64:~# hello
Hello World
root@qemux86-64:~#

```

2:

### 런타임 패키지 변수에 대해 설명하시오:

- **RRECOMMENDS :**
  - **Description:** This variable lists packages that are not essential but are recommended to be installed along with the package being defined. If these recommended packages are available, they will be installed by default.
  - **Usage:** It's used when a package can provide additional functionality if another package is present, but can still function without it.
- **RSUGGESTS :**
  - **Description:** Similar to `RRECOMMENDS`, `RSUGGESTS` lists packages that are suggested to be used with the package being defined, but are even less critical. These suggestions are typically not installed by default and are just informational.

- **Usage:** It's useful for indicating optional packages that could enhance the functionality or user experience but are not directly tied to the package's operation.
- RPROVIDES :
  - **Description:** This variable is used to specify that the package being defined provides the features or functionality of another package (or multiple packages). It's a way of saying, "Installing this package is as good as installing the listed ones."
  - **Usage:** Commonly used in situations where multiple packages can fulfill the same dependency, or when creating a virtual/meta-package that represents a group of packages.
- RCONFLICTS :
  - **Description:** RCONFLICTS is used to declare that the package being defined cannot be installed at the same time as the listed package(s). It's a way to prevent incompatible packages from being installed together.
  - **Usage:** Important for ensuring that packages that would cause problems if installed together are kept separate. For example, two packages that try to install the same file.
- RRERPLACES :
  - **Description:** This variable is used to indicate that the package being defined should replace the listed packages. It is often used in conjunction with RCONFLICTS to specify that this package not only conflicts with but also supersedes another package.
  - **Usage:** Commonly used during upgrades or when one package is meant to entirely substitute another, effectively making the old package obsolete.

## Final exercise

- This tensorflow image is based on the hello world image.
- Reference: <https://git.yoctoproject.org/meta-tensorflow/> (meta-tensorflow/BUILD.md)

## Set up

### Download tensorflow and its dependencies

```
git clone -b kirkstone git://git.yoctoproject.org/meta-tensorflow
git clone -b kirkstone git://git.openembedded.org/meta-openembedded
```

### Root directory after cloning



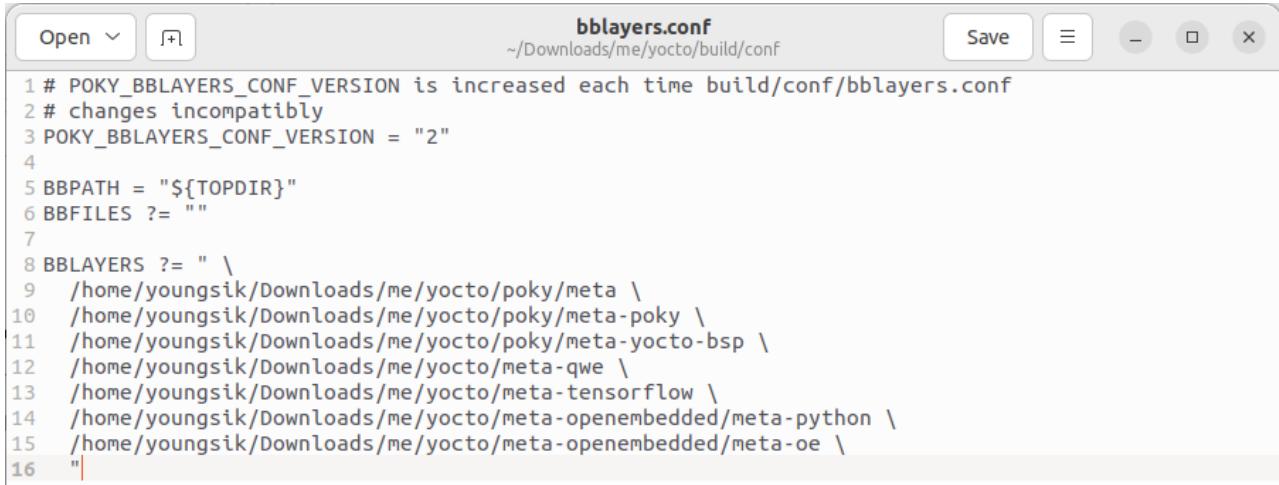
### Local.conf

- Add the tensorflow package to the final image
- Speed up the build by setting multi-threads

```
local.conf
~/Downloads/me/yocto/build/conf
Save □ ×

243 #ASSUME_PROVIDED += "libsdl2-native"
244
245 # You can also enable the Gtk UI frontend, which takes somewhat longer to build, but adds
246 # a handy set of menus for controlling the emulator.
247 #PACKAGECONFIG:append:pn-qemu-system-native = " gtk+"
248
249 #
250 # Hash Equivalence
251 #
252 # Enable support for automatically running a local hash equivalence server and
253 # instruct bitbake to use a hash equivalence aware signature generator. Hash
254 # equivalence improves reuse of sstate by detecting when a given sstate
255 # artifact can be reused as equivalent, even if the current task hash doesn't
256 # match the one that generated the artifact.
257 #
258 # A shared hash equivalent server can be set with "<HOSTNAME>:<PORT>" format
259 #
260 #BB_HASHSERVE = "auto"
261 #BB_SIGNATURE_HANDLER = "OEEquivHash"
262
263 #
264 # Memory Resident Bitbake
265 #
266 # Bitbake's server component can stay in memory after the UI for the current command
267 # has completed. This means subsequent commands can run faster since there is no need
268 # for bitbake to reload cache files and so on. Number is in seconds, after which the
269 # server will shut down.
270 #
271 #BB_SERVER_TIMEOUT = "60"
272
273 # CONF_VERSION is increased each time build/conf/ changes incompatibly and is used to
274 # track the version of this file when it was generated. This can safely be ignored if
275 # this doesn't mean anything to you.
276 CONF_VERSION = "2"
277 IMAGE_INSTALL:append = " tensorflow"
278 BB_NUMBER_THREADS = "24"
279 PARALLEL_MAKE = "-j 8"
```

## Add the layers



```
bblayers.conf
~/Downloads/me/yocto/build/conf/bblayers.conf

1 # POKY_BBLAYERS_CONF_VERSION is increased each time build/conf/bblayers.conf
2 # changes incompatibly
3 POKY_BBLAYERS_CONF_VERSION = "2"
4
5 BBPATH = "${TOPDIR}"
6 BBFILES ?= ""
7
8 BBLAYERS ?= " \
9   /home/youngsik/Downloads/me/yocto/poky/meta \
10  /home/youngsik/Downloads/me/yocto/poky/meta-poky \
11  /home/youngsik/Downloads/me/yocto/poky/meta-yocto-bsp \
12  /home/youngsik/Downloads/me/yocto/meta-qwe \
13  /home/youngsik/Downloads/me/yocto/meta-tensorflow \
14  /home/youngsik/Downloads/me/yocto/meta-openembedded/meta-python \
15  /home/youngsik/Downloads/me/yocto/meta-openembedded/meta-oe \
16 "
```

## Build image

### Set the build environment

```
source poky/oe-init-build-env ; cd ..
```

### Start build

```
bitbake -k qwe-image
```

### Verify

### Run the built image with slirp + kvm + 5GB memory:

```
runqemu qemux86-64 qwe-image slirp kvm qemuparams="-m 5120" nographic
```

## Verify the install

```
python3 -c "import tensorflow as tf;print(tf.reduce_sum(tf.random.normal([1000, 1000])))"
```

### Result

```
root@qemu86-64:~# python3 -c "import tensorflow as tf;print(tf.reduce_sum(tf.random.normal([1000, 1000])))"
2024-01-25 08:29:04.042127: I tensorflow/core/platform/cpu_feature_guard.cc:186] This TensorFlow binary is optimized with onX
To enable them in other operations, rebuild TensorFlow with the appropriate compiler flags.
2024-01-25 08:29:04.831271: I tensorflow/core/platform/cpu_feature_guard.cc:186] This TensorFlow binary is optimized with onX
To enable them in other operations, rebuild TensorFlow with the appropriate compiler flags.
tf.Tensor(-2393.77, shape=(), dtype=float32)
```

## Run tutorial case

```
cat >code-v2.py <<ENDOF
import tensorflow as tf
mnist = tf.keras.datasets.mnist

(x_train, y_train), (x_test, y_test) = mnist.load_data()
x_train, x_test = x_train / 255.0, x_test / 255.0

model = tf.keras.models.Sequential([
    tf.keras.layers.Flatten(input_shape=(28, 28)),
    tf.keras.layers.Dense(128, activation='relu'),
    tf.keras.layers.Dropout(0.2),
    tf.keras.layers.Dense(10)
])

predictions = model(x_train[:1]).numpy()
tf.nn.softmax(predictions).numpy()
loss_fn = tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True)
loss_fn(y_train[:1], predictions).numpy()

model.compile(optimizer='adam',
              loss=loss_fn,
              metrics=['accuracy'])
model.fit(x_train, y_train, epochs=5)
model.evaluate(x_test, y_test, verbose=2)

probability_model = tf.keras.Sequential([
    model,
    tf.keras.layers.Softmax()
])
probability_model(x_test[:5])

ENDOF
```

## Result

```
root@qemux86-64:~# python3 ./code-v2.py
2024-01-25 08:31:45.542195: I tensorflow/core/platform/cpu_feature_guard.cc:186] This TensorFlow binary is optimized with onX
To enable them in other operations, rebuild TensorFlow with the appropriate compiler flags.
2024-01-25 08:31:46.622067: I tensorflow/core/platform/cpu_feature_guard.cc:186] This TensorFlow binary is optimized with onX
To enable them in other operations, rebuild TensorFlow with the appropriate compiler flags.
Epoch 1/5
1875/1875 [=====] - 1s 638us/step - loss: 0.2975 - accuracy: 0.9137
Epoch 2/5
1875/1875 [=====] - 1s 658us/step - loss: 0.1422 - accuracy: 0.9576
Epoch 3/5
1875/1875 [=====] - 1s 763us/step - loss: 0.1070 - accuracy: 0.9676
Epoch 4/5
1875/1875 [=====] - 1s 688us/step - loss: 0.0881 - accuracy: 0.9730
Epoch 5/5
1875/1875 [=====] - 1s 679us/step - loss: 0.0764 - accuracy: 0.9758
313/313 - 0s - loss: 0.0775 - accuracy: 0.9752 - 204ms/epoch - 653us/step
```

## TensorFlow/TensorFlow Lite C++ Image Recognition Demo

```
time label_image.lite
```

## Result

```
root@qemux86-64:~# time label_image.lite
INFO: Loaded model /usr/share/label_image/mobilenet_v1_1.0_224_quant.tflite
INFO: resolved reporter
INFO: invoked
INFO: average time: 23.515 ms
INFO: 0.780392: 653 military uniform
INFO: 0.105882: 907 Windsor tie
INFO: 0.0156863: 458 bow tie
INFO: 0.0117647: 466 bulletproof vest
INFO: 0.00784314: 835 suit
real    0m 0.08s
user    0m 0.25s
sys     0m 0.01s
```

## Trouble-shooting

### Problem

QEMU does not support access to the internet by default.

### Solution

DNS configuration

# 자료구조, 알고리즘

## 1. Linked list 양영식



### Linked list란?

node의 모음이며, node는 data와 다른 node로 연결되는 포인터로 이루어져 있다.

### Array와 Linked list의 비교

#### Array

크기가 자주 바뀌지 않는 저장공간이 필요할 때 효율적이다.

- 원소가 연속적으로 저장되기 때문에 시작 주소를 이용해 random access가 가능하다. 즉, data access에  $O(1)$ 의 시간복잡도가 든다.
- 크기가 고정돼있기 때문에 데이터의 삽입, 삭제에  $O(n)$ 의 시간복잡도가 든다.

#### Linked list

크기가 자주 바뀌는 저장공간이 필요할 때 효율적이다.

- 원소가 불연속적으로 저장되기 때문에 random access가 불가능하다. 즉, data access에  $O(n)$ 의 시간복잡도가 든다.
- 새로운 node의 삽입과 삭제는 연결을 만드는 것과 삭제하는 것이기 때문에 데이터의 삽입, 삭제에  $O(1)$ 의 시간복잡도가 든다.
- 노드에 포인터가 저장되기 때문에 추가적인 메모리가 필요하다.

#### LinkedList.h

```
#ifndef LINKEDLIST_H
#define LINKEDLIST_H

#include <stdio.h>
#include <stdlib.h>

typedef int ElementType;
typedef struct tagNode
{
```

```

ElementType Data;
struct tagNode* NextNode;
} Node;

// 함수 원형 선언
Node* SLL_CreateNode(ElementType NewData);
void SLL_DestroyNode(Node* Node);
void SLL_AppendNode(Node** Head, Node* NewNode);
void SLL_InsertAfter(Node* Current, Node* NewNode);
void SLL_InsertNewHead(Node** Head, Node* NewHead);
void SLL_RemoveNode(Node** Head, Node* Remove);
Node* SLL_GetNodeAt(Node* Head, int Location);
int SLL_GetNodeCount(Node* Head);

#endif

```

## LinkedList.c

```

#include "LinkedList.h"

// 노드 생성
Node* SLL_CreateNode(ElementType NewData)
{
    Node* NewNode = (Node*)malloc(sizeof(Node));

    NewNode->Data = NewData;
    NewNode->NextNode = NULL;

    return NewNode;
}

// 노드 소멸
void SLL_DestroyNode(Node* Node)
{
    free(Node);
}

// 노드 추가
void SLL_AppendNode(Node** Head, Node* NewNode) // Double pointer가 필요한 이유는 head를
// call by reference하기 위해서이다!
{
    // 헤드 노드가 NULL이라면 새로운 노드가 헤드 노드가 됨
    if ((*Head) == NULL)
    {
        *Head = NewNode;
    }
    else
    {

```

```

// 테일 노드를 찾아 NewNode를 연결
Node* Tail = (*Head);
while (Tail->NextNode != NULL)
{
    Tail = Tail->NextNode;
}

Tail->NextNode = NewNode;
}

// 노드 삽입
void SLL_InsertAfter(Node* Current, Node* NewNode)
{
    NewNode->NextNode = Current->NextNode;
    Current->NextNode = NewNode;
}

void SLL_InsertNewHead(Node** Head, Node* NewHead)
{
    if ((*Head) == NULL)
    {
        (*Head) = NewHead;
    }
    else
    {
        NewHead->NextNode = (*Head);
        (*Head) = NewHead;
    }
}

// 노드 제거
void SLL_RemoveNode(Node** Head, Node* Remove)
{
    // 헤드 노드가 제거할 노드라면 헤드 노드를 다음 노드로 변경
    if (*Head == Remove)
    {
        *Head = Remove->NextNode;
    }
    else
    {
        // 제거할 노드의 이전 노드를 찾는다.
        Node* Current = *Head;
        while (Current != NULL && Current->NextNode != Remove)
        {
            Current = Current->NextNode;
        }

        if (Current != NULL)
            Current->NextNode = Remove->NextNode;
    }
    // # Memory leak 발생
}

```

```

}

// 노드 탐색
Node* SLL_GetNodeAt(Node* Head, int Location)
{
    Node* Current = Head;

    while (Current != NULL && (--Location) >= 0)
    {
        Current = Current->NextNode;
    }

    return Current;
}

// 노드 개수
int SLL_GetNodeCount(Node* Head)
{
    unsigned int Count = 0;
    Node* Current = Head;

    while (Current != NULL)
    {
        Current = Current->NextNode;
        Count++;
    }

    return Count;
}

```

### Test\_LinkedList.c

```

#include "LinkedList.h"

int main(void){
    int i=0;
    int Count=0;
    Node* List=NULL;
    Node* Current = NULL;
    Node* NewNode = NULL;

    // 노드 5개 추가
    for(i=0; i<5; i++){
        NewNode = SLL_CreateNode(i);
        SLL_AppendNode(&List, NewNode);
    }

    NewNode = SLL_CreateNode(-1);
    SLL_InsertNewHead(&List, NewNode);
}

```

```

NewNode = SLL_CreateNode(-2);
SLL_InsertNewHead(&List, NewNode);

// 리스트 출력
Count = SLL_GetNodeCount(List);
for(i=0; i<Count; i++){
    Current = SLL_GetNodeAt(List, i);
    printf("List[%d] : %d\n", i, Current->Data);
}

// 리스트의 세번째 노드 뒤에 새 노드 삽입
printf("\nInserting 3000 After [2]...\n\n");

Current = SLL_GetNodeAt(List, 2);
NewNode = SLL_CreateNode(3000);

SLL_InsertAfter(Current, NewNode);

// 리스트 출력
Count = SLL_GetNodeCount(List);
for(i=0; i<Count; i++){
    Current = SLL_GetNodeAt(List, i);
    printf("List[%d] : %d\n", i, Current->Data);
}

// 모든 노드 메모리 해제
printf("\nDestroying List...\n");

for (i=0; i<Count; i++){
    Current = SLL_GetNodeAt(List, 0);

    if(Current != NULL){
        SLL_RemoveNode(&List, Current);
        SLL_DestroyNode(Current);
    }
}

return 0;
}

```

```

List[0] : -2
List[1] : -1
List[2] : 0
List[3] : 1
List[4] : 2
List[5] : 3
List[6] : 4

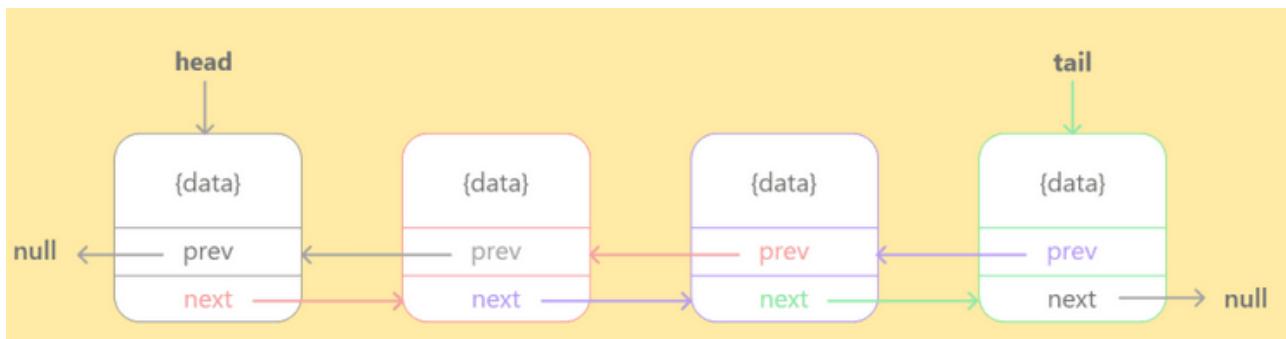
Inserting 3000 After [2]...

List[0] : -2
List[1] : -1
List[2] : 0
List[3] : 3000
List[4] : 1
List[5] : 2
List[6] : 3
List[7] : 4

Destroying List...

```

## 2. Doubly Linked List 양영식



Singly linked list는 단방향으로만 탐색이 가능하다. 이것을 개선하기 위해 node에 포인터 2개를 저장해 양방향으로 탐색할 수 있도록 만든게 Doubly linked list이다.

탐색 성능은 개선되지만 node에 2개의 포인터가 저장되기 때문에 추가적인 메모리가 필요하다.

### DoublyLinkedList.h

```

#ifndef DOUBLY_LINKED_LIST_H
#define DOUBLY_LINKED_LIST_H

#include <stdio.h>
#include <stdlib.h>

typedef int ElementType;
typedef struct tagNode
{
    ElementType Data;

```

```

struct tagNode* PrevNode;
struct tagNode* NextNode;
} Node;

// 함수 원형 선언
Node* DLL_CreateNode(ElementType NewData);
void DLL_DestroyNode(Node* Node);
void DLL_AppendNode(Node** Head, Node* NewNode);
void DLL_InsertAfter(Node* Current, Node* NewNode);
void DLL_RemoveNode(Node** Head, Node* Remove);
Node* DLL_GetNodeAt(Node* Head, int Location);
int DLL_GetNodeCount(Node* Head);

#endif

```

### DoublyLinkedList.c

```

#include "DoublyLinkedList.h"

// 노드 생성
Node* DLL_CreateNode(ElementType NewData)
{
    Node* NewNode = (Node*)malloc(sizeof(Node));

    NewNode->Data = NewData;
    NewNode->PrevNode = NULL;
    NewNode->NextNode = NULL;

    return NewNode;
}

// 노드 소멸
void DLL_DestroyNode(Node* Node)
{
    free(Node);
}

// 노드 추가
void DLL_AppendNode(Node** Head, Node* NewNode)
{
    // 헤드 노드가 NULL이라면 새로운 노드가 헤드 노드가 된다.
    if ((*Head) == NULL)
    {
        *Head = NewNode;
    }
    else
    {

```

```

// 테일을 찾아 NewNode를 연결한다.
Node* Tail = (*Head);
while (Tail->NextNode != NULL)
{
    Tail = Tail->NextNode;
}

Tail->NextNode = NewNode;
NewNode->PrevNode = Tail; // 기존 테일을 새로운 테일의 PrevNode가 가리킨다.
}

// 노드 삽입
void DLL_InsertAfter(Node* Current, Node* NewNode)
{
    NewNode->NextNode = Current->NextNode;
    NewNode->PrevNode = Current;

    if (Current->NextNode != NULL)
    {
        Current->NextNode->PrevNode = NewNode;
        Current->NextNode = NewNode;
    }
}

// 노드 제거
void DLL_RemoveNode(Node** Head, Node* Remove)
{
    if (*Head == Remove)
    {
        *Head = Remove->NextNode;
        if (*Head != NULL)
        {
            (*Head)->PrevNode = NULL;
        }
        Remove->PrevNode = NULL;
        Remove->NextNode = NULL;
    }
    else
    {
        Node* Temp = Remove;

        if (Remove->PrevNode != NULL)
        {
            Remove->PrevNode->NextNode = Temp->NextNode;
        }
        if (Remove->NextNode != NULL)
        {
            Remove->NextNode->PrevNode = Temp->PrevNode;
        }
    }

    Remove->PrevNode = NULL;
}

```

```

        Remove->NextNode = NULL;
    }
}

// 노드 탐색
Node* DLL_GetNodeAt(Node* Head, int Location)
{
    Node* Current = Head;

    while (Current != NULL && (--Location) >= 0)
    {
        Current = Current->NextNode;
    }

    return Current;
}

// 노드 개수 세기
int DLL_GetNodeCount(Node* Head)
{
    unsigned int Count = 0;
    Node* Current = Head;

    while (Current != NULL)
    {
        Current = Current->NextNode;
        Count++;
    }

    return Count;
}

void PrintNode(Node* _Node)
{
    if (_Node->PrevNode == NULL)
        printf("Prev: NULL");
    else{
        printf("Prev: %d", _Node->PrevNode->Data);
        print("Current: %d", _Node->Data);
    }

    if (_Node->NextNode == NULL)
        printf("Next: NULL");
    else
        printf("Next: %d", _Node->NextNode->Data);
}

```

**Test\_DoublyLinkedList.c**

```
#include "DoublyLinkedList.h"

int main(void)
{
    int i=0;
    int Count=0;
    Node* List=NULL;
    Node* NewNode=NULL;
    Node* Current=NULL;

    // 노드 5개 추가
    for(i=0; i<5; i++)
    {
        NewNode = DLL_CreateNode(i);
        DLL_AppendNode(&List, NewNode);
    }

    // 리스트 출력
    Count = DLL_GetNodeCount(List);
    for(i=0; i<Count; i++)
    {
        Current = DLL_GetNodeAt(List, i);
        printf("List[%d] : %d\n", i, Current->Data);
    }

    // 리스트의 세번째 노드 뒤에 새 노드 삽입
    printf("\nInserting 3000 After [2]...\n\n");

    Current = DLL_GetNodeAt(List, 2);
    NewNode = DLL_CreateNode(3000);
    DLL_InsertAfter(Current, NewNode);

    // 리스트 출력
    Count = DLL_GetNodeCount(List);
    for(i=0; i<Count; i++)
    {
        Current = DLL_GetNodeAt(List, i);
        printf("List[%d] : %d\n", i, Current->Data);
    }

    // 모든 노드를 메모리에서 제거
    printf("\nDestroying List...\n");

    for(i=0; i<Count; i++)
    {
        Current = DLL_GetNodeAt(List, 0);
        if(Current != NULL)
```

```

    {
        DLL_RemoveNode(&List, Current);
        DLL_DestroyNode(Current);
    }
}
}

```

```

List[0] : 0
List[1] : 1
List[2] : 2
List[3] : 3
List[4] : 4

Inserting 3000 After [2]...

List[0] : 0
List[1] : 1
List[2] : 2
List[3] : 3000
List[4] : 3
List[5] : 4

Destroying List...

```

### 3. Circular doubly linked list 양영식



Doubly linked list에 tail을 추가하고 head의 이전 node는 tail, tail의 다음 node는 head로 설정한 것이 Circular doubly linked list이다.

#### CircularDoublyLinkedList.h

```

#ifndef CIRCULAR_DOUBLY_LINKEDLIST_H

#define CIRCULAR_DOUBLY_LINKEDLIST_H

#include <stdio.h>
#include <stdlib.h>

typedef int ElementType;
typedef struct tagNode
{
    ElementType Data;
    struct tagNode* PrevNode;
    struct tagNode* NextNode;
} Node;

```

```
// 함수 원형 선언
Node* CDLL_CreateNode(ElementType NewData);
void CDLL_DestroyNode(Node* Node);
void CDLL_AppendNode(Node** Head, Node* NewNode);
void CDLL_InsertAfter(Node* Current, Node* NewNode);
void CDLL_RemoveNode(Node** Head, Node* Remove);
Node* CDLL_GetNodeAt(Node* Head, int Location);
int CDLL_GetNodeCount(Node* Head);

#endif
```

**CircularDoublyLinkedList.c**

```
#include "CircularDoublyLinkedList.h"

// 노드 생성
Node* CDLL_CreateNode(ElementType NewData)
{
    Node* NewNode = (Node*)malloc(sizeof(Node));

    NewNode->Data = NewData;
    NewNode->PrevNode = NULL;
    NewNode->NextNode = NULL;

    return NewNode;
}

// 노드 소멸
void CDLL_DestroyNode(Node* Node)
{
    free(Node);
}

// 노드 추가
void CDLL_AppendNode(Node** Head, Node* NewNode)
{
    // 헤드 노드가 NULL이라면 새로운 노드가 헤드 노드가 된다.
    if ((*Head) == NULL)
    {
        *Head = NewNode;
        (*Head)->NextNode = *Head;
        (*Head)->PrevNode = *Head;
    }
    else
    {
        // 테일과 헤드 사이에 NewNode를 삽입한다.
        Node* Tail = (*Head)->PrevNode;

        Tail->NextNode->PrevNode = NewNode;
```

```

    Tail->NextNode = NewNode;

    NewNode->NextNode = *Head;
    NewNode->PrevNode = Tail; // 새로운 테일의 PrevNode가 기존의 테일을 가리킨다.
}
}

// 노드 삽입
void CDLL_InsertAfter(Node* Current, Node* NewNode)
{
    NewNode->NextNode = Current->NextNode;
    NewNode->PrevNode = Current;

    if(Current->NextNode != NULL)
    {
        Current->NextNode->PrevNode = NewNode;
        Current->NextNode->PrevNode = NewNode;
    }
}

// 노드 제거
void CDLL_RemoveNode(Node** Head, Node* Remove)
{
    if (*Head == Remove)
    {
        (*Head)->PrevNode->NextNode = Remove->NextNode;
        (*Head)->NextNode->PrevNode = Remove->PrevNode;

        *Head = Remove->NextNode;

        Remove->PrevNode = NULL;
        Remove->NextNode = NULL;
    }
    else
    {
        Remove->PrevNode->NextNode = Remove->NextNode;
        Remove->NextNode->PrevNode = Remove->PrevNode;

        Remove->PrevNode = NULL;
        Remove->NextNode = NULL;
    }
}

// 노드 탐색
Node* CDLL_GetNodeAt(Node* Head, int Location)
{
    Node* Current = Head;

    while (Current != NULL && (--Location) >= 0)
    {
        Current = Current->NextNode;
    }
}

```

```

    return Current;
}

// 노드 개수
int CDLL_GetNodeCount(Node* Head)
{
    unsigned int Count = 0;
    Node* Current = Head;

    while (Current != NULL)
    {
        Current = Current->NextNode;
        Count++;

        if (Current == Head)
            break;
    }

    return Count;
}

void printNode(Node* _Node)
{
    if(_Node->NextNode == NULL)
        printf("Prev: NULL");
    else
        printf("Prev: %d", _Node->PrevNode->Data);

    printf("Current: %d", _Node->Data);

    if(_Node->NextNode == NULL)
        printf("Next: NULL");
    else
        printf("Next: %d", _Node->NextNode->Data);
}

```

### Test\_CircularDoublyLinkedList.c

```

#include "CircularDoublyLinkedList.h"

int main(void){
    int i=0;
    int Count=0;
    Node* List = NULL;
    Node* NewNode = NULL;
    Node* Current = NULL;

    // 노드 5개 추가

```

```

for(i=0; i<5; i++){
    NewNode = CDLL_CreateNode(i);
    CDLL_AppendNode(&List, NewNode);
}

// 리스트 출력
Count = CDLL_GetNodeCount(List);
for(i=0; i<Count; i++){
    Current = CDLL_GetNodeAt(List, i);
    printf("List[%d] : %d\n", i, Current->Data);
}

// 리스트의 세 번째 칸 뒤에 노드 삽입
printf("\nInserting 3000 After [2]...\n\n");

Current = CDLL_GetNodeAt(List, 2);
NewNode = CDLL_CreateNode(3000);
CDLL_InsertAfter(Current, NewNode);

printf("\nRemoving Node at 2...\n");
Current = CDLL_GetNodeAt(List, 2);
CDLL_RemoveNode(&List, Current);
CDLL_DestroyNode(Current);

// 리스트 출력
// (노드 개수의 2배만큼 루프를 돌며 환형임을 확인한다.)
Count = CDLL_GetNodeCount(List);
for(i=0; i<Count*2; i++){
    if(i==0){
        Current = List;
    }
    else{
        Current = Current->NextNode;
    }

    printf("List[%d] : %d\n", i, Current->Data);
}

// 모든 노드를 메모리에서 제거
printf("\nDestroying List...\n");
Count = CDLL_GetNodeCount(List);

for(i=0; i<Count; i++){
    Current = CDLL_GetNodeAt(List, 0);

    if(Current != NULL){
        CDLL_RemoveNode(&List, Current);
        CDLL_DestroyNode(Current);
    }
}

return 0;

```

```
}
```

```
List[0] : 0
List[1] : 1
List[2] : 2
List[3] : 3
List[4] : 4

Inserting 3000 After [2]...

Removing Node at 2...
List[0] : 0
List[1] : 1
List[2] : 3
List[3] : 4
List[4] : 0
List[5] : 1
List[6] : 3
List[7] : 4

Destroying List...
```

## 4. ArrayStack 양영식



A stack holds a linear sequence of elements where the element inserted at the last, is the first element to come out, i.e. a stack is based on **Last In, First Out (LIFO)** principle.

### Advantages

- Good for cache

### Problems of this code

- Segmentation fault might occur(can be prevented using Valgrind)

#### ArrayStack.h

```
#ifndef ARRAYSTACK_H
#define ARRAYSTACK_H

#include <stdio.h>
#include <stdlib.h>

typedef int ElementType;

typedef struct tagNode
{
    ElementType Data;
```

```

} Node;

typedef struct tagArrayStack
{
    int Capacity;
    int Top;
    Node *Nodes;
} ArrayStack;
void AS_CreateStack(ArrayStack **Stack, int Capacity);
void AS_DestroyStack(ArrayStack *Stack);
void AS_Push(ArrayStack *Stack, ElementType Data);
ElementType AS_Pop(ArrayStack *Stack);
int AS_GetSize(ArrayStack *Stack);
int AS_IsEmpty(ArrayStack *Stack);

#endif

```

### ArrayStack.c

```

#include "ArrayStack.h"

void AS_CreateStack(ArrayStack** Stack, int Capacity)
{
    // 스택을 자유 저장소에 생성
    (*Stack) = (ArrayStack*)malloc(sizeof(ArrayStack));

    // 입력된 Capacity만큼의 노드를 자유 저장소에 생성
    (*Stack)->Nodes = (Node*)malloc(sizeof(Node) * Capacity);

    // Capacity와 Top을 초기화
    (*Stack)->Capacity = Capacity;
    (*Stack)->Top = -1;
}

void AS_DestroyStack(ArrayStack* Stack)
{
    // 노드 메모리 해제
    free(Stack->Nodes);

    // 스택 메모리 해제
    free(Stack);
}

void AS_Push(ArrayStack* Stack, ElementType Data)
{
    Stack->Top++;
    Stack->Nodes[Stack->Top].Data = Data;
}

```

```

ElementType AS_Pop(ArrayStack* Stack)
{
    int Position = Stack->Top--;
    return Stack->Nodes[Position].Data;
}

ElementType AS_Top(ArrayStack* Stack)
{
    return Stack->Nodes[Stack->Top].Data;
}

int AS_GetSize(ArrayStack* Stack)
{
    return Stack->Top + 1;
}

int AS_IsEmpty(ArrayStack* Stack)
{
    return (Stack->Top == -1);
}

```

**test\_ArrayStack.h**

```

#include "ArrayStack.h"

int main(void)
{
    int i=0;
    ArrayStack* Stack;

    AS_CreateStack(&Stack, 10);

    AS_Push(Stack, 3);
    AS_Push(Stack, 37);
    AS_Push(Stack, 11);
    AS_Push(Stack, 12);

    printf("Capacity: %d, Size: %d, Top: %d\n\n", Stack->Capacity, AS_GetSize(Stack),
AS_Top(Stack));

    for(i=0; i<4; i++)
    {
        if(AS_IsEmpty(Stack))
            break;

        printf("Popped: %d, ", AS_Pop(Stack));

        if(!AS_IsEmpty(Stack))
            printf("Current Top: %d\n", AS_Top(Stack));
    }
}

```

```

    else
        printf("Stack Is Empty.\n");
    }

    AS_DestroyStack(Stack);
}

```

## 5. LinkedListStack 양영식

LinkedList로 이루어진 Stack이다.

### 결과

```

Size: 4, Top: hij
Popped: hij, Current Top: efg
Popped: efg, Current Top: def
Popped: def, Current Top: abc
Popped: abc, Stack Is Empty.
[1] + Done

```

LLS\_Pop()에서 CurrentTop != NULL && 는 필요없지않나?

### 코드

#### Test\_LinkedListStack.c

```

#include "LinkedListStack.h"

int main(void){
    int i=0;
    int Count = 0;
    Node* Popped;

    LinkedListStack* Stack;

    LLS_CreateStack(&Stack);

    LLS_Push(Stack, LLS_CreateNode("abc"));
    LLS_Push(Stack, LLS_CreateNode("def"));
    LLS_Push(Stack, LLS_CreateNode("efg"));
    LLS_Push(Stack, LLS_CreateNode("hij"));

    Count = LLS_GetSize(Stack);
    printf("Size: %d, Top: %s\n\n", Count, LLS_Top(Stack)->Data);
}

```

```

for(i=0; i<Count; i++){
    if(LLS_IsEmpty(Stack)){
        break;
    }

    Popped = LLS_Pop(Stack);
    printf("Popped: %s, ", Popped->Data);

    LLS_DestroyNode(Popped);

    if(!LLS_IsEmpty(Stack)){
        printf("Current Top: %s\n", LLS_Top(Stack)->Data);
    }
    else{
        printf("Stack Is Empty.\n");
    }
}

LLS_DestroyStack(Stack);
}

```

### LinkedListStack.h

```

#ifndef LINKEDLIST_STACK_H
#define LINKEDLIST_STACK_H

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

typedef struct tagNode
{
    char* Data;
    struct tagNode* NextNode;
} Node;

typedef struct tagLinkedListStack
{
    Node* List;
    Node* Top;
} LinkedListStack;

void LLS_CreateStack(LinkedListStack** Stack);
void LLS_DestroyStack(LinkedListStack* Stack);

Node* LLS_CreateNode(char* NewData);
void LLS_DestroyNode(Node* _Node);

```

```

void LLS_Push(LinkedListStack* Stack, Node* NewNode);
Node* LLS_Pop(LinkedListStack* Stack);

Node* LLS_Top(LinkedListStack* Stack);
int LLS_GetSize(LinkedListStack* Stack);
int LLS_IsEmpty(LinkedListStack* Stack);

#endif // LINKEDLIST_STACK_H

```

**LinkedListStack.c**

```

#include "LinkedListStack.h"

void LLS_CreateStack(LinkedListStack** Stack)
{
    (*Stack) = (LinkedListStack*)malloc(sizeof(LinkedListStack));
    (*Stack)->List = NULL;
    (*Stack)->Top = NULL;
}

void LLS_DestroyStack(LinkedListStack* Stack)
{
    while(!LLS_IsEmpty(Stack))
    {
        Node* Popped = LLS_Pop(Stack);
        LLS_DestroyNode(Popped);
    }

    free(Stack);
}

Node* LLS_CreateNode(char* NewData)
{
    Node* NewNode = (Node*)malloc(sizeof(Node));
    NewNode->Data = (char*)malloc(strlen(NewData) + 1);

    strcpy(NewNode->Data, NewData);

    NewNode->NextNode = NULL;

    return NewNode;
}

void LLS_DestroyNode(Node* _Node)
{
    free(_Node->Data);
    free(_Node);
}

```

```

void LLS_Push(LinkedListStack* Stack, Node* NewNode)
{
    if(Stack->List == NULL)
    {
        Stack->List = NewNode;
    }
    else
    {
        Stack->Top->NextNode = NewNode;
    }

    Stack->Top = NewNode;
}

Node* LLS_Pop(LinkedListStack* Stack)
{
    Node* TopNode = Stack->Top;

    if(Stack->List == Stack->Top)
    {
        Stack->List = NULL;
        Stack->Top = NULL;
    }
    else
    {
        Node* CurrentTop = Stack->List;

        while(CurrentTop != NULL && CurrentTop->NextNode != Stack->Top)
        {
            CurrentTop = CurrentTop->NextNode;
        }

        Stack->Top = CurrentTop;
        CurrentTop->NextNode = NULL;
    }

    return TopNode;
}

Node* LLS_Top(LinkedListStack* Stack)
{
    return Stack->Top;
}

int LLS_GetSize(LinkedListStack* Stack)
{
    int Count = 0;
    Node* Current = Stack->List;

    while(Current != NULL)
    {
        Current = Current->NextNode;
    }
}

```

```

        Count++;
    }

    return Count;
}

int LLS_IsEmpty(LinkedListStack* Stack)
{
    return (Stack->List == NULL);
}

```

## 6. chapter02 연습문제, 사칙연산 양영식

### 연습문제 1

4번 수행

### 연습문제 2

#### 02

```

void AS_Push(ArrayStack* Stack, ElementType Data) {
    // Check if the stack is full
    if (Stack->Top >= Stack->Capacity) {
        // Increase capacity by 30%
        int newCapacity = Stack->Capacity + (Stack->Capacity * 30 / 100);

        // Allocate new memory block with increased capacity
        Node* newNodes = realloc(Stack->Nodes, newCapacity * sizeof(Node));

        // Update the stack with new settings
        Stack->Nodes = newNodes;
        Stack->Capacity = newCapacity;
    }
    // Perform the push operation
    int Position = Stack->Top;
    Stack->Nodes[Position].Data = Data;
    Stack->Top++;
}

```

## 연습문제 3

03

```
#include <stdlib.h> // For dynamic memory allocation

ElementType AS_Pop(ArrayStack* Stack) {
    // Check if the stack is empty before attempting to pop
    if (Stack->Top == 0) {
        // Handle the empty stack case (e.g., return a default value or error)
        // Assuming ElementType is int for this example
        return -1; // Example: Returning -1 for an empty stack
    }

    // Pop the element
    int Position = --(Stack->Top);
    ElementType poppedData = Stack->Nodes[Position].Data;

    // Check if the stack usage is less than 70% of its capacity
    if (Stack->Top < (Stack->Capacity * 70 / 100)) {
        // Calculate the new capacity, ensuring it's not less than the current size
        // of the stack
        int newCapacity = Stack->Top > (Stack->Capacity * 70 / 100) ? Stack->Top :
        (Stack->Capacity * 70 / 100);

        // Reallocate memory to decrease the capacity
        Node* newNodes = realloc(Stack->Nodes, newCapacity * sizeof(Node));

        // Update the stack with new settings
        Stack->Nodes = newNodes;
        Stack->Capacity = newCapacity;
    }

    return poppedData;
}
```

## 사칙연산

Operators that should be calculated earlier should be in a higher position of the stack

**Calculator.py**

```
def infix_to_postfix(expression):
    print('\n'+expression)
```

```

# Priority of operators
priority = {'+": 1, '-': 1, '*': 2, '/': 2, '()': 0}

stack = [] # Stack for operators
postfix = []
number = ''

# Iterate over each character in the expression
for char in expression:
    if char.isdigit():
        # Build the number from consecutive digits
        number += char
    else:
        if number:
            # Append the complete number to postfix and reset number
            postfix.append(number)
            number = ''
        if char == '(':
            stack.append(char)
        elif char == ')':
            while stack and stack[-1] != '(':
                postfix.append(stack.pop())
            stack.pop() # Remove the left parenthesis
        else: # To calculate operators with higher priority!
            while stack and priority[char] <= priority[stack[-1]]:
                postfix.append(stack.pop())
            stack.append(char)
print(stack, postfix)

# Add the last number if there is one
if number:
    postfix.append(number)

# Pop any remaining operators from the stack to the output
while stack:
    postfix.append(stack.pop())

return ' '.join(postfix)

# Example
expression = "2+3*(4-5)"
postfix = infix_to_postfix(expression)
print(postfix)
expression = "(2*3+4)*5+6"
postfix = infix_to_postfix(expression)
print(postfix)
expression = "2*3+4*(5-6)"
postfix = infix_to_postfix(expression)
print(postfix)

```

**Test\_Calculator.c**

```
#include <stdio.h>
#include <string.h>
#include "Calculator.h"

int main( void )
{
    char InfixExpression[100];
    char PostfixExpression[100];

    double Result = 0.0;

    memset( InfixExpression, 0, sizeof(InfixExpression) );
    memset( PostfixExpression, 0, sizeof(PostfixExpression) );

    printf( "Enter Infix Expression:" );
    scanf( "%s", InfixExpression );

    GetPostfix( InfixExpression, PostfixExpression );

    printf( "Infix:%s\nPostfix:%s\n",
    InfixExpression,
    PostfixExpression );

    Result = Calculate( PostfixExpression );

    printf( "Calculation Result : %f\n", Result );

    return 0;
}
```

**Calculator.h**

```
#ifndef CALCULATOR_H
#define CALCULATOR_H

#include <stdio.h>
#include "LinkedListStack.h"

typedef enum{
    LEFT_PARENTHESIS='(', RIGHT_PARENTHESIS=')',
    PLUS='+', MINUS='-', MULTIPLY='*', DIVIDE='/',
    SPACE=' ', OPERAND
} SYMBOL;

int IsNumber(char Cipher);
```

```

unsigned int GetNextToken(char* Expression, char* Token, int* TYPE);
int IsPrior(char Operator1, char Operator2);
void GetPostfix(char* InfixExpression, char* PostfixExpression);
double Calculate(char* PostfixExpression);

#endif

```

**Test\_Calculator.c**

```

#include "Calculator.h"

char NUMBER[] = { '0', '1', '2', '3', '4', '5', '6', '7', '8', '9', '.' };

// The time complexity is O(n), which is inefficient. It can be done in O(1)
int IsNumber( char Cipher )
{
    int i = 0;
    int ArrayLength = sizeof(NUMBER);

    for ( i=0; i<ArrayLength; i++ )
    {
        if ( Cipher == NUMBER[i] )
            return 1;
    }

    return 0;
}

unsigned int GetNextToken( char* Expression, char* Token, int* TYPE )
{
    unsigned int i = 0;

    for ( i=0 ; 0 != Expression[i]; i++ )
    {
        Token[i] = Expression[i];

        if ( IsNumber( Expression[i] ) == 1 )
        {
            *TYPE = OPERAND;

            if ( IsNumber(Expression[i+1]) != 1 )
                break;
        }
        else
        {
            *TYPE = Expression[i];
            break;
        }
    }
}

```

```

Token[++i] = '\0'; // End of line
return i;
}

int GetPriority(char Operator, int InStack)
{
    int Priority = -1;

    switch (Operator)
    {
        case LEFT_PARENTHESIS:
            if ( InStack )
                Priority = 3;
            else
                Priority = 0;
            break;

        case MULTIPLY:
        case DIVIDE:
            Priority = 1;
            break;

        case PLUS:
        case MINUS:
            Priority = 2;
            break;
    }

    return Priority;
}

int IsPrior( char OperatorInStack, char OperatorInToken )
{
    return ( GetPriority(OperatorInStack, 1) > GetPriority(OperatorInToken, 0) );
}

void GetPostfix( char* InfixExpression, char* PostfixExpression )
{
    LinkedListStack* Stack;

    char Token[32];
    int Type = -1;
    unsigned int Position = 0;
    unsigned int Length = strlen( InfixExpression );

    LLS_CreateStack(&Stack);

    while ( Position < Length )
    {
        Position += GetNextToken( &InfixExpression[Position], Token, &Type );
}

```

```

if ( Type == OPERAND )
{
    strcat( PostfixExpression, Token );
    strcat( PostfixExpression, " " );
}
else if ( Type == RIGHT_PARENTHESIS )
{
    while ( !LLS_IsEmpty(Stack) )
    {
        Node* Popped = LLS_Pop( Stack );

        if ( Popped->Data[0] == LEFT_PARENTHESIS )
        {
            LLS_DestroyNode( Popped );
            break;
        }
        else
        {
            strcat( PostfixExpression, Popped->Data );
            LLS_DestroyNode( Popped );
        }
    }
}
else
{
    while ( !LLS_IsEmpty( Stack ) &&
           !IsPrior( LLS_Top( Stack )->Data[0], Token[0] ) )
    {
        Node* Popped = LLS_Pop( Stack );

        if ( Popped->Data[0] != LEFT_PARENTHESIS )
            strcat( PostfixExpression, Popped->Data );

        LLS_DestroyNode( Popped );
    }

    LLS_Push( Stack, LLS_CreateNode( Token ) );
}
}

while ( !LLS_IsEmpty( Stack ) )
{
    Node* Popped = LLS_Pop( Stack );

    if ( Popped->Data[0] != LEFT_PARENTHESIS )
        strcat( PostfixExpression, Popped->Data );

    LLS_DestroyNode( Popped );
}

LLS_DestroyStack(Stack);
}

```

```

double Calculate( char* PostfixExpression )
{
    LinkedListStack* Stack;
    Node* ResultNode;

    double Result;
    char Token[32];
    int Type = -1;
    unsigned int Read = 0;
    unsigned int Length = strlen( PostfixExpression );

    LLS_CreateStack(&Stack);

    while ( Read < Length )
    {
        Read += GetNextToken( &PostfixExpression[Read], Token, &Type );

        if ( Type == SPACE )
            continue;

        if ( Type == OPERAND )
        {
            Node* NewNode = LLS_CreateNode( Token );
            LLS_Push( Stack, NewNode );
        }
        else
        {
            char ResultString[32];
            double Operator1, Operator2, TempResult;
            Node* OperatorNode;

            OperatorNode = LLS_Pop( Stack );
            Operator2 = atof( OperatorNode->Data );
            LLS_DestroyNode( OperatorNode );

            OperatorNode = LLS_Pop( Stack );
            Operator1 = atof( OperatorNode->Data );
            LLS_DestroyNode( OperatorNode );

            switch (Type)
            {
                case PLUS:     TempResult = Operator1 + Operator2; break;
                case MINUS:    TempResult = Operator1 - Operator2; break;
                case MULTIPLY: TempResult = Operator1 * Operator2; break;
                case DIVIDE:   TempResult = Operator1 / Operator2; break;
            }

            gcvt( TempResult, 10, ResultString );
            LLS_Push( Stack, LLS_CreateNode( ResultString ) );
        }
    }
}

```

```

ResultNode = LLS_Pop( Stack );
Result = atof( ResultNode->Data );
LLS_DestroyNode( ResultNode );

LLS_DestroyStack( Stack );

return Result;
}

```

```

Enter Infix Expression:10*(3.14+10)/3*(2+100)
Infix:10*(3.14+10)/3*(2+100)
Postfix:10 3.14 10 +*3 /2 100 +*
Calculation Result : 4467.600000

```

### LinkedListStack.h

```

#ifndef LINKEDLIST_STACK_H
#define LINKEDLIST_STACK_H

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

typedef struct tagNode
{
    char* Data;
    struct tagNode* NextNode;
} Node;

typedef struct tagLinkedListStack
{
    Node* List;
    Node* Top;
} LinkedListStack;

void LLS_CreateStack(LinkedListStack** Stack);
void LLS_DestroyStack(LinkedListStack* Stack);

Node* LLS_CreateNode(char* NewData);
void LLS_DestroyNode(Node* _Node);

void LLS_Push(LinkedListStack* Stack, Node* NewNode);
Node* LLS_Pop(LinkedListStack* Stack);

Node* LLS_Top(LinkedListStack* Stack);

```

```

int LLS_GetSize(LinkedListStack* Stack);
int LLS_IsEmpty(LinkedListStack* Stack);

#endif // LINKEDLIST_STACK_H

```

**LinkedListStack.c**

```

#include "LinkedListStack.h"

void LLS_CreateStack(LinkedListStack** Stack)
{
    (*Stack) = (LinkedListStack*)malloc(sizeof(LinkedListStack));
    (*Stack)->List = NULL;
    (*Stack)->Top = NULL;
}

void LLS_DestroyStack(LinkedListStack* Stack)
{
    while(!LLS_IsEmpty(Stack))
    {
        Node* Popped = LLS_Pop(Stack);
        LLS_DestroyNode(Popped);
    }

    free(Stack);
}

Node* LLS_CreateNode(char* NewData)
{
    Node* NewNode = (Node*)malloc(sizeof(Node));
    NewNode->Data = (char*)malloc(strlen(NewData) + 1);

    strcpy(NewNode->Data, NewData);

    NewNode->NextNode = NULL;

    return NewNode;
}

void LLS_DestroyNode(Node* _Node)
{
    free(_Node->Data);
    free(_Node);
}

void LLS_Push(LinkedListStack* Stack, Node* NewNode)
{
    if(Stack->List == NULL)
    {

```

```

        Stack->List = NewNode;
    }
else
{
    Stack->Top->NextNode = NewNode;
}

Stack->Top = NewNode;
}

Node* LLS_Pop(LinkedListStack* Stack)
{
    Node* TopNode = Stack->Top;

    if(Stack->List == Stack->Top)
    {
        Stack->List = NULL;
        Stack->Top = NULL;
    }
    else
    {
        Node* CurrentTop = Stack->List;

        while(CurrentTop != NULL && CurrentTop->NextNode != Stack->Top)
        {
            CurrentTop = CurrentTop->NextNode;
        }

        Stack->Top = CurrentTop;
        CurrentTop->NextNode = NULL;
    }

    return TopNode;
}

Node* LLS_Top(LinkedListStack* Stack)
{
    return Stack->Top;
}

int LLS_GetSize(LinkedListStack* Stack)
{
    int Count = 0;
    Node* Current = Stack->List;

    while(Current != NULL)
    {
        Current = Current->NextNode;
        Count++;
    }

    return Count;
}

```

```

    }

int LLS_IsEmpty(LinkedListStack* Stack)
{
    return (Stack->List == NULL);
}

```

## 7. Circular queue 양영식

### CircularQueue.h

```

#ifndef CIRCULAR_QUEUE_H
#define CIRCULAR_QUEUE_H

#include <stdio.h>
#include <stdlib.h>

typedef int ElementType;

typedef struct tagNode
{
    ElementType Data;
} Node;

typedef struct tagCircularQueue
{
    int Capacity;
    int Front;
    int Rear;

    Node *Nodes;
} CircularQueue;

void CQ_CreateQueue(CircularQueue **Queue, int Capacity);
void CQ_DestroyQueue(CircularQueue *Queue);
void CQ.Enqueue(CircularQueue *Queue, ElementType Data);
ElementType CQ.Dequeue(CircularQueue *Queue);
int CQ.GetSize(CircularQueue *Queue);
int CQ.IsEmpty(CircularQueue *Queue);
int CQ.IsFull(CircularQueue *Queue);

#endif

```

**CircularQueue.c**

```
#include "CircularQueue.h"

void CQ_CreateQueue( CircularQueue** Queue, int Capacity)
{
    (*Queue) = ( CircularQueue*)malloc(sizeof( CircularQueue ));

    // 입력된 Capacity+1 만큼의 노드를 자유 저장소에 생성
    (*Queue)->Nodes = (Node*)malloc(sizeof(Node )* ( Capacity+1 ) );

    (*Queue)->Capacity = Capacity;
    (*Queue)->Front = 0;
    (*Queue)->Rear = 0;
}

void CQ_DestroyQueue( CircularQueue* Queue )
{
    free(Queue->Nodes);
    free(Queue );
}

void CQ.Enqueue( CircularQueue* Queue, ElementType Data)
{
    int Position=0;

    if(Queue->Rear==Queue->Capacity)
    {
        Position=Queue->Rear;
        Queue->Rear=0;
    }
    else
        Position=Queue->Rear++;

    Queue->Nodes[Position].Data=Data;
}

ElementType CQ.Dequeue( CircularQueue* Queue )
{
    int Position = Queue->Front;

    if ( Queue->Front == Queue->Capacity )
        Queue->Front = 0;
    else
        Queue->Front++;

    return Queue->Nodes[Position].Data;
}

int CQ.GetSize( CircularQueue* Queue )
```

```

{
    if ( Queue->Front <= Queue->Rear )
        return Queue->Rear - Queue->Front;
    else
        return Queue->Rear + (Queue->Capacity - Queue->Front) + 1;
}

int CQ_IsEmpty( CircularQueue* Queue )
{
    return (Queue->Front == Queue->Rear);
}

int CQ_IsFull( CircularQueue* Queue )
{
    if ( Queue->Front < Queue->Rear )
        return ( Queue->Rear - Queue->Front ) == Queue->Capacity;
    else
        return ( Queue->Rear + 1 ) == Queue->Front;
}

```

### Test\_CircularQueue.c

```

#include "CircularQueue.h"

int main( void )
{
    int i;
    CircularQueue* Queue;

    CQ_CreateQueue(&Queue, 10);

    CQ.Enqueue( Queue, 1 );
    CQ.Enqueue( Queue, 2 );
    CQ.Enqueue( Queue, 3 );
    CQ.Enqueue( Queue, 4 );

    for ( i=0; i<3; i++ )
    {
        printf( "Dequeue: %d, ", CQ.Dequeue( Queue ) );
        printf( "Front:%d, Rear:%d\n", Queue->Front, Queue->Rear );
    }

    i = 100;
    while ( CQ_IsFull( Queue ) == 0 )
    {
        CQ.Enqueue( Queue, i++ );
    }

    printf( "Capacity: %d, Size: %d\n\n",

```

```

    Queue->Capacity, CQ_GetSize( Queue ) );

while ( CQ_IsEmpty( Queue ) == 0 )
{
    printf( "Dequeue: %d, ", CQ_Dequeue( Queue ) );
    printf( "Front:%d, Rear:%d\n", Queue->Front, Queue->Rear );
}

CQ_DestroyQueue( Queue );

return 0;
}

```

## 8. LinkedQueue 양영식

### Test\_LinkedQueue.c

```

#include "LinkedQueue.h"

int main( void )
{
    Node* Popped;
    LinkedQueue* Queue;

    LQ_CreateQueue( &Queue );

    LQ.Enqueue( Queue, LQ_CreateNode( "abc" ) );
    LQ.Enqueue( Queue, LQ_CreateNode( "def" ) );
    LQ.Enqueue( Queue, LQ_CreateNode( "efg" ) );
    LQ.Enqueue( Queue, LQ_CreateNode( "hij" ) );

    printf( "Queue Size: %d\n", Queue->Count );

    while ( LQ_IsEmpty(Queue) == 0 )
    {
        Popped = LQ_Dequeue( Queue );
        printf( "Dequeue: %s \n", Popped->Data );
        LQ_DestroyNode( Popped );
    }

    LQ_DestroyQueue( Queue );
}

```

**LinkedQueue.h**

```
#ifndef LINKED_QUEUE_H
#define LINKED_QUEUE_H

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

typedef struct tagNode
{
    char* Data;
    struct tagNode* NextNode;
} Node;

typedef struct tagLinkedQueue
{
    Node* Front;
    Node* Rear;
    int Count;
} LinkedQueue;

void LQ_CreateQueue( LinkedQueue** Queue );
void LQ_DestroyQueue( LinkedQueue* Queue );

Node* LQ_CreateNode( char* NewData );
void LQ_DestroyNode( Node* _Node );

void LQ.Enqueue( LinkedQueue* Queue, Node* NewNode );
Node* LQ.Dequeue( LinkedQueue* Queue );

int LQ_IsEmpty( LinkedQueue* Queue );

#endif
```

**LinkedQueue.c**

```
#include "LinkedQueue.h"

void LQ_CreateQueue( LinkedQueue** Queue )
{
    (*Queue) = (LinkedQueue*)malloc(sizeof(LinkedQueue));
    (*Queue)->Front = NULL;
    (*Queue)->Rear = NULL;
    (*Queue)->Count = 0;
}
```

```

void LQ_DestroyQueue( LinkedQueue* Queue )
{
    while ( !LQ_IsEmpty(Queue) )
    {
        Node* Popped = LQ_Dequeue(Queue);
        LQ_DestroyNode( Popped );
    }

    free( Queue );
}

Node* LQ_CreateNode( char* NewData )
{
    Node* NewNode = (Node*)malloc(sizeof(Node));
    NewNode->Data = (char*)malloc(strlen(NewData) + 1);

    strcpy(NewNode->Data, NewData);

    NewNode->NextNode = NULL;

    return NewNode;
}

void LQ_DestroyNode( Node* _Node )
{
    free( _Node->Data );
    free( _Node );
}

void LQ_Enqueue( LinkedQueue* Queue, Node* NewNode )
{
    if ( Queue->Front == NULL )
    {
        Queue->Front = NewNode;
        Queue->Rear = NewNode;
        Queue->Count++;
    }
    else
    {
        Queue->Rear->NextNode = NewNode;
        Queue->Rear = NewNode;
        Queue->Count++;
    }
}

Node* LQ_Dequeue( LinkedQueue* Queue )
{
    Node* Front = Queue->Front;

    if ( Queue->Front->NextNode == NULL )
    {
        Queue->Front = NULL;
    }
}

```

```

        Queue->Rear = NULL;
    }
else
{
    Queue->Front = Queue->Front->NextNode;
}

Queue->Count--;

return Front;
}

int LQ_IsEmpty( LinkedQueue* Queue )
{
    return (Queue->Front == NULL);
}

```

## 10. 트리(LCRS, Binary) 양영식

<https://github.com/vacu9708/Data-structure/tree/main/Tree>

연습문제

1.

4번 (두번째 레벨에 빈 node가 있다)

2.

pre: +\*12-78 | in: 1\*2 + 7-8 | post: 12\*78-+

3.

1개 (A의 parent를 D로 혹은 D의 parent를 A로)

## 11. Expression Tree 양영식

<https://github.com/vacu9708/Data-structure/blob/main/Expression%20tree/readme.md>

## 12. Union-Find

<https://github.com/vacu9708/Data-structure/tree/main/Disjoint%20set%2C%20Union-Find>

## 13. Sort

Bubble sort (<https://github.com/vacu9708/Algorithm/tree/main/Sorting%20algorithm/Bubble%20sort>)

```

#include <stdio.h>

void BubbleSort(int DataSet[], int Length){
    int i = 0;

```

```

int j = 0;
int temp = 0;

for(i=0; i<Length-1; i++){
    for(j=0; j<Length-(i+1); j++){
        if(DataSet[j] > DataSet[j+1]){
            temp = DataSet[j+1];
            DataSet[j+1] = DataSet[j];
            DataSet[j] = temp;
        }
    }
}

int main(void){
    int DataSet[] = {6, 4, 2, 3, 1, 5};
    int Length = sizeof DataSet / sizeof DataSet[0];
    int i = 0;

    BubbleSort(DataSet, Length);

    for(i=0; i<Length; i++){
        printf("%d ", DataSet[i]);
    }

    printf("\n");

    return 0;
}

```

Insertion sort(<https://github.com/vacu9708/Algorithm/tree/main/Sorting%20algorithm/Insertion%20sort>)

```

#include <stdio.h>
#include <string.h>

void InsertionSort(int DataSet[], int Length)
{
    int i=0, j=0, value=0;

    for(i=1; i<Length; i++)
    {
        if(DataSet[i-1] <= DataSet[i])
            continue;

        value = DataSet[i];

        for(j=0; j<i; j++)
        {
            if(DataSet[j] > value)
            {
                memmove(&DataSet[j+1], &DataSet[j], sizeof(DataSet[0])*(i-j));
                DataSet[j] = value;
            }
        }
    }
}

```

```

        break;
    }
}
}

int main(void)
{
    int DataSet[] = {6, 4, 2, 3, 1, 5};
    int Length = sizeof DataSet / sizeof DataSet[0];
    int i = 0;

    InsertionSort(DataSet, Length);

    for(i=0; i<Length; i++)
        printf("%d ", DataSet[i]);

    printf("\n");

    return 0;
}

```

quick sort(<https://github.com/vacu9708/Algorithm/blob/main/Sorting%20algorithm/Quick%20sort/README.md>)

### QuickSort1.c

```

#include <stdio.h>

void Swap(int *a, int *b) {
    int temp = *a;
    *a = *b;
    *b = temp;
}

int Partition(int DataSet[], int Left, int Right){
    int First = Left;
    int Pivot = DataSet[First];

    ++Left;

    while(Left <= Right){
        while(DataSet[Left] <= Pivot && Left < Right){
            ++Left;
        }
        while(DataSet[Right] > Pivot && Left <= Right){
            --Right;
        }
    }
}

```

```

    }

    if(Left < Right){
        Swap(&DataSet[Left], &DataSet[Right]);
    }
    else{
        break;
    }
}

Swap(&DataSet[First], &DataSet[Right]);
return Right;
}

void QuickSort(int DataSet[], int Left, int Right){
    if(Left < Right){
        int Index = Partition(DataSet, Left, Right);

        QuickSort(DataSet, Left, Index - 1);
        QuickSort(DataSet, Index + 1, Right);
    }
}

int main(void){
    int DataSet[] = {6, 4, 2, 3, 1, 5};
    int Length = sizeof DataSet / sizeof DataSet[0];
    int i = 0;

    QuickSort(DataSet, 0, Length - 1);

    for(i = 0; i < Length; ++i){
        printf("%"d ", DataSet[i]);
    }
    printf("\\n");

    return 0;
}

```

실행결과

1 2 3 4 5 6

### QuickSort2.c

```

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

int ComparePoint(const void *_elem1, const void *_elem2)
{

```

```

int *elem1 = (int *)_elem1;
int *elem2 = (int *)_elem2;

if(*elem1 > *elem2)
    return 1;
else if(*elem1 < *elem2)
    return -1;
else
    return 0;
}

int main(void)
{
    int DataSet[] = {6, 4, 2, 3, 1, 5};
    int Length = sizeof DataSet / sizeof DataSet[0];
    int i = 0;

    qsort(DataSet, Length, sizeof(int), ComparePoint);

    for(i = 0; i < Length; i++)
        printf("%"d " , DataSet[i]);

    printf("\\n");
}

```

실행결과

1 2 3 4 5 6

## 17. Priority queue

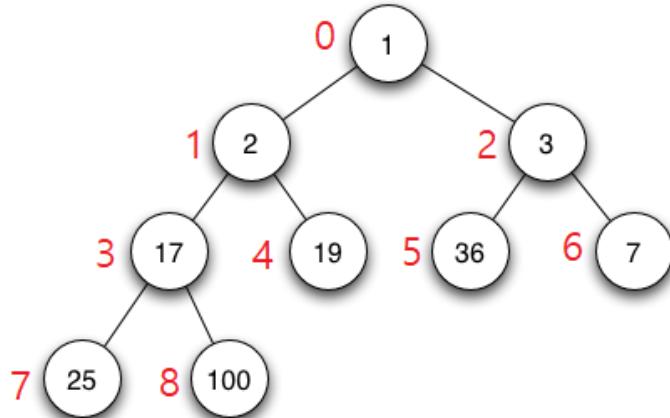
### Priority queue

A priority queue is a collection of data elements similar to a regular queue in which an element with the highest priority leaves first.

Elements with higher priority are placed before their children

The element with the highest priority is always at the beginning of the queue. Normally **Heap** is used to implement a priority queue.

## Example of Min heap



## Why is heap used instead of linked list to implement priority queue?

- **Heap**
  - `heap_push()` : **O(logn)** [Upward-heapifying takes **O(logn)** as it is the reverse process of the binary search]
  - `heap_pop()`: **O(logn)** [Downward-heapifying takes **O(logn)** in the same way as the binary search]
- **Linked list**
  - `insert()` : **O(n)** [It takes **O(n)** to compare the priority of all the elements before inserting)]
  - `get()` : **O(1)** [Because all that needs to be done is just get the front data]

As stated above, priority queue using heap is more stable than linked list in terms of time complexity while priority queue using linked list is unstable.

## Caution

Before heapifying, the array must first be transformed into a heap tree by lifting up elements with higher priority, which takes  $O(N\log N)$

```

#include <iostream>
#include <vector>
using namespace std;

void print_array(vector<int>& heap_tree) {
    int tree_size = heap_tree.size();
    for (int i = 0; i < tree_size; ++i)
        cout << heap_tree[i] << " ";
  
```

```

        cout << "\n";
    }

void downward_heapify(vector<int>& heap_tree, int curr) { // For pop()
    int left = curr*2+1, right = curr*2+2; // children
    int least = curr;
    // If a child is smaller than its parent, swap them
    if (left < heap_tree.size() && heap_tree[left] < heap_tree[least]) // becomes
max_heapify by changing the inequality
        least=left;
    if (right < heap_tree.size() && heap_tree[right] < heap_tree[least])
        least=right;
    if (least != curr) {
        swap(heap_tree[least], heap_tree[curr]);
        downward_heapify(heap_tree, least);
    }
}

void upward_heapify(vector<int>& heap_tree, int curr){ // For push()
    int parent=(curr-1)/2;
    // Return if out of range or the parent is smaller than the current node
    if (!(curr>=1 || heap_tree[curr]<heap_tree[parent]))
        return;
    swap(heap_tree[curr], heap_tree[parent]);
    upward_heapify(heap_tree, parent);
}

// void heapify_all(vector<int>& heap_tree){
//     for (int i = (heap_tree.size()-2)/2; i >= 0; i--) // from last child's parent
//         downward_heapify(heap_tree, i);
// }

void heap_push(vector<int>& heap_tree, int data) {
    heap_tree.push_back(data);
    upward_heapify(heap_tree, heap_tree.size() - 1);
}

int heap_pop(vector<int>& heap_tree) {
    int highest_priority = heap_tree[0];
    // Move the the last node to the place to be deleted for a better time complexity
in the vector
    heap_tree[0] = heap_tree[heap_tree.size() - 1];
    heap_tree.pop_back();
    downward_heapify(heap_tree, 0);
    return highest_priority;
}

int main() {
    vector<int> heap_tree;
    for (int i = 5; i > 0; i--)
        heap_push(heap_tree, i);
    printf("Heap tree : "); print_array(heap_tree);
    for(int i=0; i<5; i++){
}

```

```

        printf("Get (%d)\n", heap_pop(heap_tree));
        printf("Heap tree : "); print_array(heap_tree);
    }
}

```

## chapter 06 탐색

### 14. BinarySearch

```

typedef struct tagPoint
{
    int id;
    double point;
} Point;

Point DataSet[]=
{
    // ID, 구매포인트
    (Point){1, 100.0},
    (Point){2, 200.0},
    (Point){3, 300.0},
    (Point){4, 400.0},
    (Point){5, 500.0},
    (Point){6, 600.0},
    (Point){7, 700.0},
    (Point){8, 800.0},
    (Point){9, 900.0},
};

```

```

#include <stdio.h>
#include <stdlib.h>
#include "Point.h"

Point* BinarySearch(Point PointList[], int Size, double Target)
{
    int Left, Right, Mid;

    Left = 0;
    Right = Size - 1;

    while (Left <= Right)
    {
        Mid = (Left + Right) / 2;

        if (Target == PointList[Mid].point)
            return &PointList[Mid];
    }
}

```

```

        else if (Target > PointList[Mid].point)
            Left = Mid + 1;
        else
            Right = Mid - 1;
    }

    return NULL;
}

int ComparePoint(const void *_elem1, const void *_elem2)
{
    Point *elem1 = (Point *)_elem1;
    Point *elem2 = (Point *)_elem2;

    if (elem1->point > elem2->point)
        return 1;
    else if (elem1->point < elem2->point)
        return -1;
    else
        return 0;
}

int main(void)
{
    int Length = sizeof(DataSet) / sizeof(DataSet[0]);
    Point* found = NULL;

    qsort(DataSet, Length, sizeof(Point), ComparePoint);

    found = BinarySearch(DataSet, Length, 500.0);

    printf("found... ID: %d, Point: %f\n", found->id, found->point);
}

```

## 15. BinarySearchTree

```

#ifndef BINARY_SEARCH_TREE_H
#define BINARY_SEARCH_TREE_H

#include <stdio.h>
#include <stdlib.h>

typedef int ElementType;

typedef struct tagBSTNode
{
    ElementType Data;
    struct tagBSTNode *Left;
    struct tagBSTNode *Right;
}

```

```

} BSTNode;

BSTNode* BST_CreateNode(ElementType NewData);
void BST_DestroyNode(BSTNode* Node);
void BST_DestroyTree(BSTNode* Tree);

BSTNode* BST_SearchNode(BSTNode* Tree, ElementType Target);
BSTNode* BST_SearchMinNode(BSTNode* Tree);
void BST_InsertNode(BSTNode* Tree, BSTNode* Child);
BSTNode* BST_RemoveNode(BSTNode* Tree, BSTNode* Parent, ElementType Target);
void BST_InorderPrintTree(BSTNode* Node);

#endif

```

```

#include "BinarySearchTree.h"

BSTNode* BST_CreateNode(ElementType NewData)
{
    BSTNode* NewNode = (BSTNode*)malloc(sizeof(BSTNode));

    NewNode->Left = NULL;
    NewNode->Right = NULL;
    NewNode->Data = NewData;

    return NewNode;
}

void BST_DestroyNode(BSTNode* Node)
{
    free(Node);
}

void BST_DestroyTree(BSTNode* Tree)
{
    if (Tree->Right != NULL)
        BST_DestroyTree(Tree->Right);

    if (Tree->Left != NULL)
        BST_DestroyTree(Tree->Left);

    Tree->Left = NULL;
    Tree->Right = NULL;

    BST_DestroyNode(Tree);
}

BSTNode* BST_SearchNode(BSTNode* Tree, ElementType Target)
{
    if (Tree == NULL) return NULL;

    if (Tree->Data > Target)

```

```

        return BST_SearchNode(Tree->Left, Target);
    else if (Tree->Data < Target)
        return BST_SearchNode(Tree->Right, Target);
    else
        return Tree;
}

BSTNode* BST_SearchMinNode(BSTNode* Tree)
{
    if (Tree == NULL) return NULL;

    if (Tree->Left == NULL)
        return Tree;
    else
        return BST_SearchMinNode(Tree->Left);
}

void BST_InsertNode(BSTNode* Tree, BSTNode* Child)
{
    if (Tree->Data < Child->Data)
    {
        if (Tree->Right == NULL)
            Tree->Right = Child;
        else
            BST_InsertNode(Tree->Right, Child);
    }
    else if (Tree->Data > Child->Data)
    {
        if (Tree->Left == NULL)
            Tree->Left = Child;
        else
            BST_InsertNode(Tree->Left, Child);
    }
}

BSTNode* BST_RemoveNode(BSTNode* Tree, BSTNode* Parent, ElementType Target)
{
    BSTNode* RemovedNode = NULL;

    if (Tree == NULL) return NULL;

    if (Tree->Data > Target)
        RemovedNode = BST_RemoveNode(Tree->Left, Tree, Target);
    else if (Tree->Data < Target)
        RemovedNode = BST_RemoveNode(Tree->Right, Tree, Target);
    else
    {
        RemovedNode = Tree;

        if (Tree->Left == NULL && Tree->Right == NULL)
        {
            if (Parent->Left == Tree)
                Parent->Left = NULL;
        }
    }
}

```

```

        else
            Parent->Right = NULL;
    }
    else
    {
        if (Tree->Left != NULL && Tree->Right != NULL)
        {
            BSTNode* MinNode = BST_SearchMinNode(Tree->Right);
            MinNode = BST_RemoveNode(Tree, NULL, MinNode->Data);
            Tree->Data = MinNode->Data;
        }
        else
        {
            BSTNode* Temp = NULL;

            if (Tree->Left != NULL)
                Temp = Tree->Left;
            else
                Temp = Tree->Right;

            if (Parent->Left == Tree)
                Parent->Left = Temp;
            else
                Parent->Right = Temp;
        }
    }
}

return RemovedNode;
}

void BST_InorderPrintTree(BSTNode* Node)
{
    if (Node == NULL) return;

    BST_InorderPrintTree(Node->Left);
    printf("%d ", Node->Data);
    BST_InorderPrintTree(Node->Right);
}

```

```

#include "BinarySearchTree.h"

void PrintSearchResult(int SearchTarget, BSTNode* Result)
{
    if (Result != NULL)
        printf("Found : %d \n", Result->Data);
    else
        printf("Not Found : %d \n", SearchTarget);
}

int main(void)

```

```

{
    // 노드생성
    BSTNode* Tree = BST_CreateNode(123);
    BSTNode* Node = NULL;

    // 노드 추가
    BST_InsertNode(Tree, BST_CreateNode(22));
    BST_InsertNode(Tree, BST_CreateNode(9918));
    BST_InsertNode(Tree, BST_CreateNode(424));
    BST_InsertNode(Tree, BST_CreateNode(17));
    BST_InsertNode(Tree, BST_CreateNode(3));

    BST_InsertNode(Tree, BST_CreateNode(98));
    BST_InsertNode(Tree, BST_CreateNode(34));

    BST_InsertNode(Tree, BST_CreateNode(760));
    BST_InsertNode(Tree, BST_CreateNode(317));
    BST_InsertNode(Tree, BST_CreateNode(1));

    int SearchTarget = 17;
    Node = BST_SearchNode(Tree, SearchTarget);
    PrintSearchResult(SearchTarget, Node);

    SearchTarget = 117;
    Node = BST_SearchNode(Tree, SearchTarget);
    PrintSearchResult(SearchTarget, Node);

    // 트리출력
    BST_InorderPrintTree(Tree);
    printf("\n");

    // 노드제거
    Node = BST_RemoveNode(Tree, NULL, 98);
    BST_DestroyNode(Node);

    // 트리출력
    BST_InorderPrintTree(Tree);
    printf("\n");

    printf("Inserting 111...\n");

    BST_InsertNode(Tree, BST_CreateNode(111));
    BST_InorderPrintTree(Tree);
    printf("\n");

    BST_DestroyNode(Tree);
}

```

## 16. RedBlackTree



A Red-Black Tree is a type of self-balancing binary search tree.

It stays balanced with operations like insertion, deletion, and search in  $O(\log n)$  time complexity.

### Red black tree properties

The properties that must be preserved are:

- **Node Color:** Every node is either red or black.
- **Root Property:** The root of the tree is always black.
- **Leaf Property:** Every leaf (NIL node) is black.
- **Red Node Property:** Red nodes cannot have red children (i.e., no two red nodes can be adjacent).
- **Black Depth Property:** Every path from a node to its descendant NIL nodes has the same number of black nodes.

### Cases During Insertion

When a new node is inserted, it is initially colored `red`. This can lead to a violation of the Red-Black Tree properties, particularly the Red Node Property.

The cases during insertion are:

- **Case 1:** The new node is at the root of the tree.
  - **Action:** Color the node black to satisfy the root property.
- **Case 2:** The new node's parent is black.
  - **Action:** No action needed, as the tree remains valid.
- **Case 3:** The new node's parent and uncle are red.
  - **Action:** Color both the parent and the uncle black and the grandparent red. Then, recheck the tree starting from the grandparent.
- **Case 4:** The new node's parent is red but the uncle is black; the new node is added to the right of the left child or to the left of the right child (the "triangle" configuration).
  - **Action:** Perform a rotation (left or right, respectively) on the parent, transforming the case into Case 5.
- **Case 5:** The new node's parent is red but the uncle is black; the new node is added to the left of the left child or to the right of the right child (the "line" configuration).
  - **Action:** Perform a rotation on the grandparent (right or left, respectively), swap the colors of the grandparent and parent, and recheck the tree.

### Cases During Deletion

Deletion can be more complex due to the need to replace the deleted node and potentially rebalance the tree. The cases during deletion are:

- **Case 1:** The sibling of the node being fixed is red.
  - **Action:** Recolor the sibling and the parent, and perform a rotation on the parent.
- **Case 2:** The sibling and its children are black.
  - **Action:** Recolor the sibling and move the problem up the tree by rechecking the tree starting from the parent.
- **Case 3:** The sibling is black, its left child is red, and its right child is black.
  - **Action:** Recolor the sibling and its left child, and perform a rotation on the sibling. This transforms the case into Case 4.
- **Case 4:** The sibling is black and its right child is red.
  - **Action:** Perform a rotation on the parent, recolor the sibling and the right child, and recheck the tree.

```
#ifndef REDBLACKTREE_H
#define REDBLACKTREE_H

#include <stdio.h>
#include <stdlib.h>

typedef int ElementType;

typedef struct tagRBTNode
{
    struct tagRBTNode* Parent;
    struct tagRBTNode* Left;
    struct tagRBTNode* Right;

    enum {RED, BLACK} Color;
    ElementType Data;
} RBTNode;

void RBT_DestroyTree(RBTNode* Tree);

RBTNode* RBT_CreateNode(ElementType NewData);
void RBT_DestroyNode(RBTNode* Node);

RBTNode* RBT_SearchNode(RBTNode* Tree, ElementType Target);
RBTNode* RBT_SearchMinNode(RBTNode* Tree);
void RBT_InsertNode(RBTNode** Tree, RBTNode* NewNode);
void RBT_InsertNodeHelper(RBTNode** Tree, RBTNode* NewNode);
RBTNode* RBT_RemoveNode(RBTNode** Root, ElementType Target);
void RBT_RebuildAfterInsert(RBTNode** Tree, RBTNode* NewNode);
void RBT_RebuildAfterRemove(RBTNode** Tree, RBTNode* X);

void RBT_PrintTree(RBTNode* Node, int Depth, int BlackCount);
void RBT_RotateLeft(RBTNode** Root, RBTNode* Parent);
void RBT_RotateRight(RBTNode** Root, RBTNode* Parent);

#endif
```

```

#include "RedBlackTree.h"
#include <stdio.h>

extern RBTNode* Nil;

RBTNode* RBT_CreateNode(ElementType NewData)
{
    RBTNode* NewNode = (RBTNode*)malloc(sizeof(RBTNode));

    NewNode->Parent = NULL;
    NewNode->Left = NULL;
    NewNode->Right = NULL;
    NewNode->Data = NewData;
    NewNode->Color = BLACK;

    return NewNode;
}

void RBT_DestroyNode(RBTNode* Node)
{
    free(Node);
}

void RBT_DestroyTree(RBTNode* Tree)
{
    if (Tree->Right != Nil)
        RBT_DestroyTree(Tree->Right);

    if (Tree->Left != Nil)
        RBT_DestroyTree(Tree->Left);

    Tree->Left = NULL;
    Tree->Right = NULL;

    if (Tree != Nil)
        RBT_DestroyNode(Tree);
}

RBTNode* RBT_SearchNode(RBTNode* Tree, ElementType Target)
{
    if (Tree == Nil)
        return NULL;

    if (Tree->Data > Target)
        return RBT_SearchNode(Tree->Left, Target);
    else if (Tree->Data < Target)
        return RBT_SearchNode(Tree->Right, Target);
    else
        return Tree;
}

RBTNode* RBT_SearchMinNode(RBTNode* Tree)

```

```

{
    if (Tree == Nil)
        return Nil;

    if (Tree->Left == Nil)
        return Tree;
    else
        return RBT_SearchMinNode(Tree->Left);
}

void RBT_InsertNode(RBTNode** Tree, RBTNode* NewNode)
{
    RBT_InsertNodeHelper(Tree, NewNode);

    NewNode->Color = RED;
    NewNode->Left = Nil;
    NewNode->Right = Nil;

    RBT_RebuildAfterInsert(Tree, NewNode);
}

void RBT_InsertNodeHelper(RBTNode** Tree, RBTNode* NewNode)
{
    if (*Tree == NULL)
        *Tree = NewNode;

    if ((*Tree)->Data < NewNode->Data)
    {
        if ((*Tree)->Right == Nil)
        {
            (*Tree)->Right = NewNode;
            NewNode->Parent = *Tree;
        }
        else
            RBT_InsertNodeHelper(&((*Tree)->Right), NewNode);
    }
    else if ((*Tree)->Data > NewNode->Data)
    {
        if ((*Tree)->Left == Nil)
        {
            (*Tree)->Left = NewNode;
            NewNode->Parent = *Tree;
        }
        else
            RBT_InsertNodeHelper(&((*Tree)->Left), NewNode);
    }
}

void RBT_RotateRight(RBTNode** Root, RBTNode* Parent)
{
    RBTNode* LeftChild = Parent->Left;

    Parent->Left = LeftChild->Right;
}

```

```

if (LeftChild->Right != Nil)
    LeftChild->Right->Parent = Parent;

LeftChild->Parent = Parent->Parent;

if (Parent->Parent == NULL)
    (*Root) = LeftChild;
else
{
    if (Parent == Parent->Parent->Left)
        Parent->Parent->Left = LeftChild;
    else
        Parent->Parent->Right = LeftChild;
}

LeftChild->Right = Parent;
Parent->Parent = LeftChild;
}

void RBT_RotateLeft(RBTNode** Root, RBTNode* Parent)
{
    RBTNode* RightChild = Parent->Right;

    Parent->Right = RightChild->Left;

    if (RightChild->Left != Nil)
        RightChild->Left->Parent = Parent;

    RightChild->Parent = Parent->Parent;

    if (Parent->Parent == NULL)
        (*Root) = RightChild;
    else
    {
        if (Parent == Parent->Parent->Left)
            Parent->Parent->Left = RightChild;
        else
            Parent->Parent->Right = RightChild;
    }

    RightChild->Left = Parent;
    Parent->Parent = RightChild;
}

void RBT_RebuildAfterInsert(RBTNode** Root, RBTNode* X)
{
    while (X != *Root && X->Parent->Color == RED)
    {
        if (X->Parent == X->Parent->Parent->Left)
        {
            RBTNode* Uncle = X->Parent->Parent->Right;
}

```

```

if (Uncle->Color == RED)
{
    X->Parent->Color = BLACK;
    Uncle->Color = BLACK;
    X->Parent->Parent->Color = RED;

    X = X->Parent->Parent;
}
else
{
    if (X == X->Parent->Right)
    {
        X = X->Parent;
        RBT_RotateLeft(Root, X);
    }

    X->Parent->Color = BLACK;
    X->Parent->Parent->Color = RED;

    RBT_RotateRight(Root, X->Parent->Parent);
}
}
else
{
    RBTNode* Uncle = X->Parent->Parent->Left;

    if (Uncle->Color == RED)
    {
        X->Parent->Color = BLACK;
        Uncle->Color = BLACK;
        X->Parent->Parent->Color = RED;

        X = X->Parent->Parent;
    }
    else
    {
        if (X == X->Parent->Left)
        {
            X = X->Parent;
            RBT_RotateRight(Root, X);
        }

        X->Parent->Color = BLACK;
        X->Parent->Parent->Color = RED;

        RBT_RotateLeft(Root, X->Parent->Parent);
    }
}
}

(*Root)->Color = BLACK;
}

```

```

RBTNode* RBT_RemoveNode(RBTNode** Root, ElementType Data)
{
    RBTNode* Removed = NULL;
    RBTNode* Successor = NULL;
    RBTNode* Target = RBT_SearchNode(*Root, Data);

    if (Target == NULL)
        return NULL;

    if (Target->Left == Nil || Target->Right == Nil)
    {
        Removed = Target;
    }
    else
    {
        Removed = RBT_SearchMinNode(Target->Right);
        Target->Data = Removed->Data;
    }

    if (Removed->Left != Nil)
        Successor = Removed->Left;
    else
        Successor = Removed->Right;

    Successor->Parent = Removed->Parent;

    if (Removed->Parent == NULL)
        (*Root) = Successor;
    else
    {
        if (Removed == Removed->Parent->Left)
            Removed->Parent->Left = Successor;
        else
            Removed->Parent->Right = Successor;
    }

    if (Removed->Color == BLACK)
        RBT_RebuildAfterRemove(Root, Successor);

    return Removed;
}

void RBT_RebuildAfterRemove(RBTNode** Root, RBTNode* Successor)
{
    RBTNode* Sibling = NULL;

    while (Successor != *Root && Successor->Color == BLACK)
    {
        if (Successor == Successor->Parent->Left)
        {
            Sibling = Successor->Parent->Right;

            if (Sibling->Color == RED)

```

```

    {
        Sibling->Color = BLACK;
        Successor->Parent->Color = RED;
        RBT_RotateLeft(Root, Successor->Parent);
    }
    else
    {
        if (Sibling->Left->Color == BLACK && Sibling->Right->Color == BLACK)
        {
            Sibling->Color = RED;
            Successor = Successor->Parent;
        }
        else
        {
            if (Sibling->Right->Color == RED)
            {
                Sibling->Left->Color = BLACK;
                Sibling->Color = RED;
                RBT_RotateRight(Root, Sibling);
                Sibling = Successor->Parent->Right;
            }

            Sibling->Color = Successor->Parent->Color;
            Successor->Parent->Color = BLACK;
            Sibling->Right->Color = BLACK;
            RBT_RotateLeft(Root, Successor->Parent);

            Successor = *Root;
        }
    }
}
else
{
    Sibling = Successor->Parent->Left;

    if (Sibling->Color == RED)
    {
        Sibling->Color = BLACK;
        Successor->Parent->Color = RED;
        RBT_RotateRight(Root, Successor->Parent);
    }
    else
    {
        if (Sibling->Left->Color == BLACK && Sibling->Right->Color == BLACK)
        {
            Sibling->Color = RED;
            Successor = Successor->Parent;
        }
        else
        {
            if (Sibling->Left->Color == RED)
            {
                Sibling->Right->Color = BLACK;
            }
        }
    }
}

```

```

        Sibling->Color = RED;
        RBT_RotateLeft(Root, Sibling);
        Sibling = Successor->Parent->Left;
    }

    Sibling->Color = Successor->Parent->Color;
    Successor->Parent->Color = BLACK;
    Sibling->Left->Color = BLACK;
    RBT_RotateRight(Root, Successor->Parent);

    Successor = *Root;
}
}

}

}

Successor->Color = BLACK;
}

void RBT_PrintTree(RBTNode* Node, int Depth, int BlackCount)
{
    int i = 0;
    char c = 'X';
    char v = -1;
    char cnt[100];

    if(Node==NULL || Node==Nil)
        return;

    if(Node->Color == BLACK)
        BlackCount++;

    if(Node->Parent != NULL)
    {
        v = Node->Parent->Data;

        if(Node->Parent->Left == Node)
            c = 'L';
        else
            c = 'R';
    }

    if (Node->Left == Nil && Node->Right == Nil)
        sprintf(cnt, "---- %d", BlackCount);
    else
        strncpy(cnt, "", sizeof(cnt));

    for (i = 0; i < Depth; i++)
        printf(" ");

    printf("%d %s [%c, %d] %s\n", Node->Data, (Node->Color == RED) ? "RED" : "BLACK",
c, v, cnt);

    RBT_PrintTree(Node->Left, Depth + 1, BlackCount);
}

```

```
RBT_PrintTree(Node->Right, Depth + 1, BlackCount);
}
```

```
#include "RedBlackTree.h"

RBTNode *Nil;

int main(void)
{
    RBTNode* Tree = NULL;
    RBTNode* Node = NULL;

    Nil = RBT_CreateNode(0);
    Nil->Color = BLACK;

    while(1)
    {
        int cmd = 0;
        int param = 0;
        char buffer[10];

        printf("Enter command number : \n");
        printf("(1) Create a node, (2) Remove a node (3) Search a node\n");
        printf("(4) Display Tree (5) quit\n");
        printf("Command : number:");

        fgets(buffer, sizeof(buffer)-1, stdin);
        sscanf(buffer, "%d", &cmd);

        if(cmd < 1 || cmd > 5)
        {
            printf("Invalid command number.\n");
            continue;
        }
        else if ( cmd == 4)
        {
            RBT_PrintTree(Tree, 0, 0);
            printf("\n");
            continue;
        }
        else if ( cmd == 5)
        {
            break;
        }

        printf("Enter parameter (1-200): \n");

        fgets(buffer, sizeof(buffer)-1, stdin);
        sscanf(buffer, "%d", &param);

        if(param < 1 || param > 200)
```

```
{  
    printf("Invalid parameter.\n");  
    continue;  
}  
  
switch(cmd)  
{  
    case 1:  
        RBT_InsertNode(&Tree, RBT_CreateNode(param));  
        break;  
    case 2:  
        Node = RBT_RemoveNode(&Tree, param);  
  
        if(Node == NULL)  
        {  
            printf("Not found node to delete %d\n", param);  
        }  
        else  
        {  
            RBT_DestroyNode(Node);  
        }  
        break;  
    case 3:  
        Node = RBT_SearchNode(Tree, param);  
  
        if(Node == NULL)  
        {  
            printf("Not found node %d\n", param);  
        }  
        else  
        {  
            printf("Found node %d(Color:%s)\n", Node->Data, (Node->Color ==  
RED) ? "RED" : "BLACK");  
        }  
        break;  
    }  
  
    printf("\n");  
}  
  
RBT_DestroyTree(Tree);  
return 0;  
}
```

## Chapter 8 해시테이블

### 18. Simple hash table

#### Hash table



A hashmap, also known as a hash table or a dictionary, is an associative array that can map keys to values. A hashmap uses a hash function to compute an index, also known as a hash code, into an array of buckets or slots, from which the desired value can be found.

#### Key components

- **Keys:** The unique identifiers used to access corresponding values in the map. Each key is unique within a hashmap.
- **Values:** The data or information that is associated with a key.
- **Hash function:** A function that converts keys into array indices. A good hash function is essential for distributing entries across the hashmap's buckets evenly.
- **Buckets or Slots:** The positions in the array where the key-value pairs are stored. The size of the array defines the capacity of the hashmap.

#### Characteristics

- **Efficiency:** In the best case, the complexity of search, insert, and delete operations are  $O(1)$ , thanks to the direct index access. However, in the worst case (e.g., when many keys collide), these operations can degrade to  $O(n)$ , where  $n$  is the number of keys.
- **Unordered:** Hashmaps do not maintain the order of their elements, so they are not suitable for applications where the order of elements is important.
- **Dynamic Resizing:** Many hashmaps automatically resize themselves when their load factor (a measure of how full the hashmap is) exceeds a certain threshold, to reduce hash collisions.

#### Functionalities

- **Insertion (Put):** Adds a new key-value pair to the hashmap. If the key already exists, its value is updated.
- **Deletion (Remove):** Removes the key-value pair from the hashmap, if present.
- **Lookup (Get):** Returns the value associated with a key. If the key is not found, it may return a special value indicating absence (e.g., null or undefined).
- **Size:** Returns the number of key-value pairs stored in the hashmap.

## Hash collision



A hash collision occurs when two distinct keys (inputs) to a hash function produce the same output (hash value).

### Why hash collisions occur

Hash collisions are inherent to the nature of hash functions because the set of possible keys is typically much larger than the set of possible hash values.

- For example, if a hash function produces a 32-bit output, there can be at most  $2^{32}$  unique hash values. Some keys must produce the same hash value, leading to collisions.

## Handling Hash Collisions

There are several strategies for dealing with collisions:

- **Chaining:** Each bucket in the hash table points to a list (or a chain) of entries. If a collision occurs, the new key-value pair is added to the list corresponding to that bucket. Lookup operations may require a linear search through the list in the worst case.
- **Open Addressing:** All elements are stored within the array itself, and the collision resolution scheme probes the array for empty slots. Several probing techniques exist, such as:
  - **linear probing** (where the table is searched sequentially)
  - **quadratic probing** (where the interval between probes increases quadratically)
  - **double hashing** (where a second hash function determines the probe sequence).
- When a collision occurs, the hash table looks for the next available slot according to the probing sequence and stores the key-value pair there.

## Rehashing

### Why Rehashing is Necessary

- **Performance Maintenance:** As more elements are added to a hash table, collisions become more likely, and the efficiency of hash table operations can degrade. Rehashing reduces the load factor and collision frequency, maintaining operation efficiency.
- **Dynamic Scaling:** Rehashing allows hash tables to adjust their size dynamically based on the number of elements stored, accommodating both growth and, in some implementations, shrinking to save memory when many elements are removed.

### Process of Rehashing

- **Create a New Hash Table:** A new, larger hash table is created, often doubling the size of the old table to ensure that the load factor is significantly reduced after rehashing. The size is usually chosen to be

a prime number or a power of two, depending on the hash function being used, to help reduce the likelihood of collisions.

- **Recompute Hashes:** Each key in the original hash table is rehashed according to the new table's size. Because the hash function often depends on the size of the table, changing the table size means that the index at which a key-value pair is stored will likely change.
- **Reinsert Elements:** Each key-value pair from the original table is inserted into the new table based on its newly computed hash. This step effectively redistributes the elements across the new table, potentially in a more evenly spaced manner, which helps in reducing collision-related issues.
- **Dispose of the Old Table:** Once all elements have been moved to the new table, the old table is disposed of.

### SimpleHashTable.h

```
#ifndef SimpleHashTable_H
#define SimpleHashTable_H

#include <stdio.h>
#include <stdlib.h>

typedef int KeyType;
typedef int ValueType;

typedef struct tagNode
{
    KeyType Key;
    ValueType Value;
} Node;

typedef struct tagHashTable
{
    int TableSize;
    Node* Table;
} HashTable;

HashTable* SHT_CreateHashTable(int TableSize);
void SHT_Set(HashTable* HT, KeyType Key, ValueType Value);
ValueType SHT_Get(HashTable* HT, KeyType Key);
void SHT_DestroyHashTable(HashTable* HT);
int SHT_Hash(KeyType Key, int TableSize);

#endif
```

### SimpleHashTable.c

```
#include "SimpleHashTable.h"
```

```

HashTable* SHT_CreateHashTable(int TableSize)
{
    HashTable* HT = (HashTable*)malloc(sizeof(HashTable));
    HT->Table = (Node*)malloc(sizeof(Node) * TableSize);
    HT->TableSize = TableSize;

    return HT;
}

void SHT_Set(HashTable* HT, KeyType Key, ValueType Value)
{
    int Address = SHT_Hash(Key, HT->TableSize);
    HT->Table[Address].Key = Key;
    HT->Table[Address].Value = Value;
}

ValueType SHT_Get(HashTable* HT, KeyType Key)
{
    int Address = SHT_Hash(Key, HT->TableSize);
    return HT->Table[Address].Value;
}

void SHT_DestroyHashTable(HashTable* HT)
{
    free(HT->Table);
    free(HT);
}

int SHT_Hash(KeyType Key, int TableSize)
{
    return Key % TableSize;
}

```

### Test\_SimpleHashTable.c

```

#include "SimpleHashTable.h"

int main(void)
{
    HashTable* HT = SHT_CreateHashTable(193);

    SHT_Set(HT, 418, 32114);
    SHT_Set(HT, 9, 514);
    SHT_Set(HT, 27, 8917);
    SHT_Set(HT, 1031, 216);

    printf("Key: %d, Value: %d\n", 418, SHT_Get(HT, 418));
    printf("Key: %d, Value: %d\n", 9, SHT_Get(HT, 9));
}

```

```

printf("Key: %d, Value: %d\n", 27, SHT_Get(HT, 27));
printf("Key: %d, Value: %d\n", 1031, SHT_Get(HT, 1031));

SHT_DestroyHashTable(HT);
}

```

## 19. Chaining

### Solution of hash collision

#### Chaining



Chaining is a technique used to resolve hash collisions in a hash table. When two keys hash to the same index in the table and a collision occurs, chaining allows both key-value pairs to be stored at that index by linking them together in a list.

#### Implementation

- **Bucket with a linked list:** Each slot (or bucket) in the hash table array points to a linked list (or chain) of entries that have been hashed to the same index.
- **Insertion:** When inserting a new key-value pair, the hash function calculates the index for the key.
  - If the bucket at that index is empty, the key-value pair is added and becomes the first element of the list.
  - If the bucket is not empty (indicating a collision), the new key-value pair is added to the head or tail of the linked list at that index, depending on the implementation.
- **Lookup (Retrieval):** To retrieve the value associated with a specific key, the hash function is used to find the appropriate bucket. Then, a search is performed through the linked list at that index to find the key. If the key is found, the corresponding value is returned. Otherwise, a signal that the key does not exist in the table is returned.
- **Deletion:** To remove a key-value pair, the process is similar to lookup. The list is searched for the key. If found, the key-value pair is removed from the list.

#### Advantages

- **Simplicity:** Chaining is straightforward to implement and understand.
- **Flexibility:** It can handle an arbitrary number of collisions gracefully, as the linked list can grow as needed.

## Disadvantages

- **Memory Overhead:** Each entry in the list requires additional memory for the pointer to the next element, which can be significant, especially if the keys and values are small.
- **Variable Performance:** The worst-case time complexity can degrade to O(n) (where n is the number of elements) if too many keys hash to the same index, making all operations slower as they require linear search through the list.

## Optimizing Chaining

To optimize the performance of a hash table using chaining:

- A **low load factor** (the ratio of the number of stored entries to the number of buckets) can be maintained by increasing the size of the hash table as it fills up, thus spreading out the entries more evenly.
- A **good hash function** that minimizes collisions and distributes keys uniformly across the buckets is crucial.

### Chaining.h

```
#ifndef CHAINING_H
#define CHAINING_H

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

typedef char* KeyType;
typedef char* ValueType;

typedef struct tagNode
{
    KeyType Key;
    ValueType Value;

    struct tagNode* Next;
} Node;

typedef Node* List;

typedef struct tagHashTable
{
    int TableSize;
    List* Table;
} HashTable;

HashTable* CHT_CreateHashTable(int TableSize);
```

```

void CHT_DestroyHashTable(HashTable* HT);

Node* CHT_CreateNode(KeyType Key, ValueType Value);
void CHT_DestroyNode(Node* TheNode);

void CHT_Set(HashTable* HT, KeyType Key, ValueType Value);
ValueType CHT_Get(HashTable* HT, KeyType Key);
int CHT_Hash(KeyType Key, int KeyLength, int TableSize);

#endif

```

## Chaining.c

```

#include "Chaining.h"

HashTable* CHT_CreateHashTable(int TableSize)
{
    HashTable* HT = (HashTable*)malloc(sizeof(HashTable));
    HT->Table = (List*)malloc(sizeof(List) * TableSize);
    memset(HT->Table, 0, sizeof(List) * TableSize);
    HT->TableSize = TableSize;

    return HT;
}

Node* CHT_CreateNode(KeyType Key, ValueType Value)
{
    Node* NewNode = (Node*)malloc(sizeof(Node));

    NewNode->Key = (char*)malloc(sizeof(char) * (strlen(Key) + 1));
    strcpy(NewNode->Key, Key);

    NewNode->Value = (char*)malloc(sizeof(char) * (strlen(Value) + 1));
    strcpy(NewNode->Value, Value);
    NewNode->Next = NULL;

    return NewNode;
}

void CHT_DestroyNode(Node* TheNode)
{
    free(TheNode->Key);
    free(TheNode->Value);
    free(TheNode);
}

void CHT_Set(HashTable* HT, KeyType Key, ValueType Value)
{
    int Address = CHT_Hash(Key, strlen(Key), HT->TableSize);
}

```

```

Node* NewNode = CHT_CreateNode(Key, Value);

if (HT->Table[Address] == NULL)
{
    HT->Table[Address] = NewNode;
}
else
{
    List L = HT->Table[Address];
    NewNode->Next = L;
    HT->Table[Address] = NewNode;

    printf("Collision occurred : Key(%s), Address(%d)\n", Key, Address);
}
}

ValueType CHT_Get(HashTable* HT, KeyType Key)
{
    int Address = CHT_Hash(Key, strlen(Key), HT->TableSize);

    List TheList = HT->Table[Address];
    List Target = NULL;

    if (TheList == NULL)
    {
        return NULL;
    }

    while(1)
    {
        if (strcmp(TheList->Key, Key) == 0)
        {
            Target = TheList;
            break;
        }

        if (TheList->Next == NULL)
            break;
        else
            TheList = TheList->Next;
    }
    return Target->Value;
}

void CHT_DestroyList(List L)
{
    if (L == NULL)
    {
        return;
    }

    if (L->Next != NULL)

```

```

    {
        CHT_DestroyList(L->Next);
    }

    CHT_DestroyNode(L);
}

void CHT_DestroyHashTable(HashTable* HT)
{
    int i=0;
    for (i=0; i<HT->TableSize; ++i)
    {
        List L = HT->Table[i];
        CHT_DestroyList(L);
    }

    free(HT->Table);
    free(HT);
}

int CHT_Hash(KeyType Key, int KeyLength, int TableSize)
{
    int i=0;
    int HashValue = 0;

    for (i=0; i<KeyLength; ++i)
    {
        HashValue = (HashValue << 3) + Key[i];
    }

    HashValue = HashValue % TableSize;

    return HashValue;
}

```

### Test\_Chaining.c

```

#include "Chaining.h"

int main(void)
{
    HashTable* HT = CHT_CreateHashTable(12289);

    CHT_Set(HT, "MSFT", "Microsoft Corporation");
    CHT_Set(HT, "JAVA", "Sun Microsystems");
    CHT_Set(HT, "REDH", "Red Hat Linux");
    CHT_Set(HT, "APAC", "Apache Org");
    CHT_Set(HT, "ZYMZZ", "Unisys Ops Check");
}

```

```

printf("Key:%s, Value:%s\n", "MSFT", CHT_Get(HT, "MSFT"));
printf("Key:%s, Value:%s\n", "JAVA", CHT_Get(HT, "JAVA"));
printf("Key:%s, Value:%s\n", "REDH", CHT_Get(HT, "REDH"));
printf("Key:%s, Value:%s\n", "APAC", CHT_Get(HT, "APAC"));
printf("Key:%s, Value:%s\n", "ZYMZZ", CHT_Get(HT, "ZYMZZ"));

CHT_DestroyHashTable(HT);

return 0;
}

```

**Result**

```

Collision occurred : Key(ZYMZZ), Address(2120)
Key:MSFT, Value:Microsoft Corporation
Key:JAVA, Value:Sun Microsystems
Key:REDH, Value:Red Hat Linux
Key:APAC, Value:Apache Org
Key:ZYMZZ, Value:Unisys Ops Check

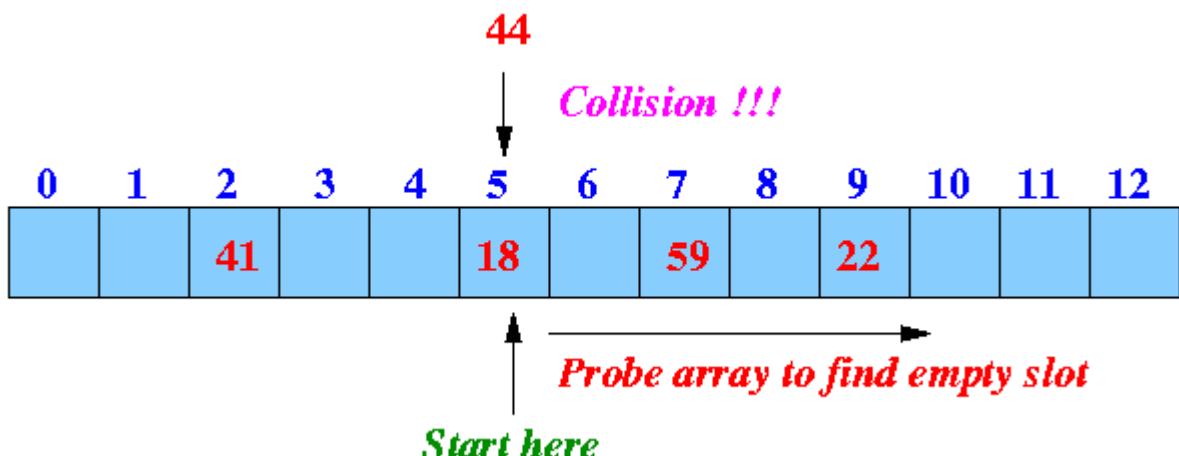
```

## 20. Open addressing

Open addressing is a collision resolution method used in hash tables when multiple keys hash to the same index.

Unlike chaining, open addressing seeks an alternative empty slot within the hash table array itself for placing the collided key-value pair.

### Key Techniques in Open Addressing



- **Linear Probing:** When a collision occurs, linear probing searches for the next available slot by moving sequentially through the table from the point of collision.
  - While simple, linear probing can lead to clustering, where blocks of consecutive slots get filled, increasing the likelihood of collisions and decreasing performance.

- **Quadratic Probing:** To address the clustering issue seen in linear probing, quadratic probing increases the interval between probes quadratically (1, 4, 9, 16, ...).
  - This spreads out the keys more evenly but can still suffer from a different form of clustering called secondary clustering because of the inherent regularity of the step size like the Linear probing.
- **Double Hashing:** Double hashing uses a second hash function to determine the step size for each probe after a collision. This method is more effective at distributing keys evenly and minimizing clustering, as the step size is unique for each key.

## Process of Open Addressing

- **Insertion:** Compute the hash index for the key. If the slot at the computed index is empty, insert the key-value pair there. If the slot is occupied (collision), use the probing method to find an empty slot and insert the key-value pair there.
- **Search/Lookup:** Compute the hash index for the key. Check the slot at the computed index; if it matches the key, return the value. If it does not match, use the probing method to find the next slot until the key is found or an empty slot is encountered, which indicates that the key is not in the table.
- **Deletion:** Mark deleted slots with a special flag ("deleted" or "tombstone") rather than leaving them empty, indicating that probing should continue past these slots.

## Advantages of Open Addressing

- **Space Efficiency:** Open addressing does not require additional data structures for storing key-value pairs outside the array, making it more space-efficient, especially when the load factor(ratio of number of elements to table size) is low to moderate.
- **Cache Efficiency:** Because all elements are stored in a contiguous block of memory, open addressing can exhibit better cache performance, leading to faster access times in some scenarios.

## Disadvantages of Open Addressing

- **Load Factor Limitations:** As the table fills up, performance can degrade significantly due to an increase in collisions and probing lengths. To maintain performance, open addressing requires keeping the load factor low, which can lead to underutilized memory.

### OpenAddressing.h

```
#ifndef OPEN_ADDRESSING_H
#define OPEN_ADDRESSING_H

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

```

typedef char *KeyType;
typedef char *ValueType;

enum ElementStatus
{
    EMPTY = 0,
    OCCUPIED = 1
};

typedef struct tagElementType
{
    KeyType Key;
    ValueType Value;
    enum ElementStatus Status;
} ElementType;

typedef struct tagHashTable
{
    int OccupiedCount;
    int TableSize;
    ElementType *Table;
} HashTable;

HashTable* OAHT_CreateHashTable(int TableSize);
void OAHT_DestroyHashTable(HashTable* HT);
void OAHT_ClearElement(ElementType* Element);

void OAHT_Set(HashTable** HT, KeyType Key, ValueType Value);
ValueType OAHT_Get(HashTable* HT, KeyType Key);
int OAHT_Hash(KeyType Key, int KeyLength, int TableSize);
int OAHT_Hash2(KeyType Key, int KeyLength, int TableSize);

void OAHT_Rehash(HashTable** HT);

#endif

```

### OpenAddressing.c

```

#include "OpenAddressing.h"

HashTable* OAHT_CreateHashTable(int TableSize) {
    HashTable* HT = (HashTable*)malloc(sizeof(HashTable));
    HT->Table = (ElementType*)malloc(sizeof(ElementType) * TableSize);

    memset(HT->Table, 0, sizeof(ElementType) * TableSize);

    HT->TableSize = TableSize;
    HT->OccupiedCount = 0;
}

```

```

    return HT;
}

void OAHT_Set(HashTable** HT, char* Key, char* Value) {
    int KeyLen, Address, StepSize;
    double Usage;

    Usage = (double)(*HT)->OccupiedCount / (double)(*HT)->TableSize;

    // If the usage is over 50%, rehashing is needed.
    if(Usage>0.5)
    {
        OAHT_Rehash(HT);
    }

    KeyLen = strlen(Key);
    Address = OAHT_Hash(Key, KeyLen, (*HT)->TableSize);
    StepSize = OAHT_Hash2(Key, KeyLen, (*HT)->TableSize);
    // Open addressing
    while((*HT)->Table[Address].Status != EMPTY &&
           strcmp((*HT)->Table[Address].Key, Key) != 0)
    {
        printf("Collision occurred! : Key : %s, Address : %d, StepSize(%d)\n", Key,
Address, StepSize);
        Address = (Address + StepSize) % (*HT)->TableSize;
    }
    // Set the key-value pair
    (*HT)->Table[Address].Key = (char*)malloc(sizeof(char) * (KeyLen + 1));
    strcpy((*HT)->Table[Address].Key, Key);
    (*HT)->Table[Address].Value = (char*)malloc(sizeof(char) * (strlen(Value) + 1));
    strcpy((*HT)->Table[Address].Value, Value);

    (*HT)->Table[Address].Status = OCCUPIED;
    (*HT)->OccupiedCount++;
    printf("Key(%s) entered at address %d\n", Key, Address);
}

ValueType OAHT_Get(HashTable* HT, KeyType Key) {
    int KeyLen = strlen(Key);
    int Address = OAHT_Hash(Key, KeyLen, HT->TableSize);
    int StepSize = OAHT_Hash2(Key, KeyLen, HT->TableSize);
    // Open addressing
    while(HT->Table[Address].Status != EMPTY &&
           strcmp(HT->Table[Address].Key, Key) != 0)
    {
        Address = (Address + StepSize) % HT->TableSize;
    }

    return HT->Table[Address].Value;
}

void OAHT_ClearElement(ElementType* Element) {

```

```

if(Element->Status == EMPTY)
    return;
```

```

free(Element->Key);
free(Element->Value);
}
```

```

void OAHT_DestroyHashTable(HashTable* HT) {
    int i = 0;
    for(i=0; i<HT->TableSize; i++)
    {
        OAHT_ClearElement(&(HT->Table[i]));
    }

    free(HT->Table);
    free(HT);
}

int OAHT_Hash(KeyType Key, int KeyLen, int TableSize) {
    int i = 0;
    int HashValue = 0;

    for(i=0; i<KeyLen; i++)
    {
        HashValue = (HashValue << 3) + Key[i];
    }

    HashValue = HashValue % TableSize;

    return HashValue;
}
// Second hash for open addressing
int OAHT_Hash2(KeyType Key, int KeyLen, int TableSize) {
    int i = 0;
    int HashValue = 0;

    for(i=0; i<KeyLen; i++)
    {
        HashValue = (HashValue << 2) + Key[i];
    }

    HashValue = HashValue % (TableSize - 3);

    return HashValue + 1;
}
// For extending the table size
void OAHT_Rehash(HashTable** HT) {
    int i = 0;
    ElementType* OldTable = (*HT)->Table;

    HashTable* NewHT = OAHT_CreateHashTable((*HT)->TableSize * 2);
}

```

```

printf("\nRehashed. New Table Size : %d\n", NewHT->TableSize);

for(i=0; i<(*HT)->TableSize; i++)
{
    if(OldTable[i].Status == OCCUPIED)
    {
        OAHT_Set(&NewHT, OldTable[i].Key, OldTable[i].Value);
    }
}

OAHT_DestroyHashTable(*HT);

*HT = NewHT;
}

```

### Test\_OpenAddressing.c

```

#include "OpenAddressing.h"

int main(void)
{
    HashTable* HT = OAHT_CreateHashTable(10);

    OAHT_Set(&HT, "MSFT", "Microsoft Corporation");
    OAHT_Set(&HT, "JAVA", "Sun Microsystems");
    OAHT_Set(&HT, "REDH", "Red Hat Linux");
    OAHT_Set(&HT, "APAC", "Apache Org");
    OAHT_Set(&HT, "ZYMZZ", "Unisys Ops Check");

    printf("Key(MSFT) : %s\n", OAHT_Get(HT, "MSFT"));
    printf("Key(JAVA) : %s\n", OAHT_Get(HT, "JAVA"));
    printf("Key(REDH) : %s\n", OAHT_Get(HT, "REDH"));
    printf("Key(APAC) : %s\n", OAHT_Get(HT, "APAC"));
    printf("Key(ZYMZZ) : %s\n", OAHT_Get(HT, "ZYMZZ"));

    OAHT_DestroyHashTable(HT);

    return 0;
}

```

## Result

```

Key(MSFT) entered at address 0
Key(JAVA) entered at address 1
Key(REDH) entered at address 6
Key(APAC) entered at address 7
Collision occurred! : Key : ZYMZZ, Address : 6, StepSize(4)
Collision occurred! : Key : ZYMZZ, Address : 0, StepSize(4)
Key(ZYMZZ) entered at address 4
Key(MSFT) : Microsoft Corporation
Key(JAVA) : Sun Microsystems
Key(REDH) : Red Hat Linux
Key(APAC) : Apache Org
Key(ZYMZZ) : Unisys Ops Check

```

# 컴퓨터구조

## 01 컴퓨터 구조 시작하기

### 컴퓨터 구조의 큰 그림

#### 컴퓨터가 이해하는 정보

컴퓨터는 0과 1로 표현된 정보만 처리할 수 있다. 0과 1로 표현되는 정보에는 크게 **data**와 **instruction**이 있다.

- 컴퓨터가 이해하는 숫자, 문자, 이미지, 동영상과 같은 정적인 정보를 가리켜 **data**라고 한다.
- 데이터를 이용해 컴퓨터를 작동시키는 정보를 **instruction**이라고 한다.

#### Example

Add 1, 2에서 Add는 instruction이고 1과 2는 data이다.

#### 컴퓨터의 4가지 핵심 부품 (자세한 설명은 뒤에서)

##### 메모리(메인 메모리)

메모리는 명령어와 데이터를 저장하는 부품이다. address로 메모리의 특정 위치에 접근할 수 있다.

현재 실행중인 프로그램의 instruction과 data가 저장된다.

책에선 메모리의 저장방식이 단순하게 표현돼있지만 실제로는 다른 방식으로 저장된다(효율적인 공간 사용을 위해 virtual memory에 저장)

## CPU

CPU는 메모리에 저장된 명령어를 읽어 들이고, 해석하고, 실행하는 장치다.

CPU의 구성 요소 중 가장 중요한 세 가지는 Arithmetic Logic Unit, Register, Control Unit이다.

- ALU: 계산을 수행
- Register: 컴퓨터에서 가장 빠른 임시저장장치이다. (flip flop)
- Control Unit: control signal을 보낸다. (메모리 읽기, 쓰기, 명령 실행 등)

## 보조기억장치

### 메인 메모리의 단점

- 메모리는 데이터를 영구저장할 수 없다.(DRAM은 refresh 필요, SRAM도 전원 유지 필수)
- 가격당 용량이 작다(비싸다)

보조기억장치는 메인메모리보다 느리지만 이러한 단점이 없다.

HDD, SSD, USB 등이 대표적인 보조기억장치이다.

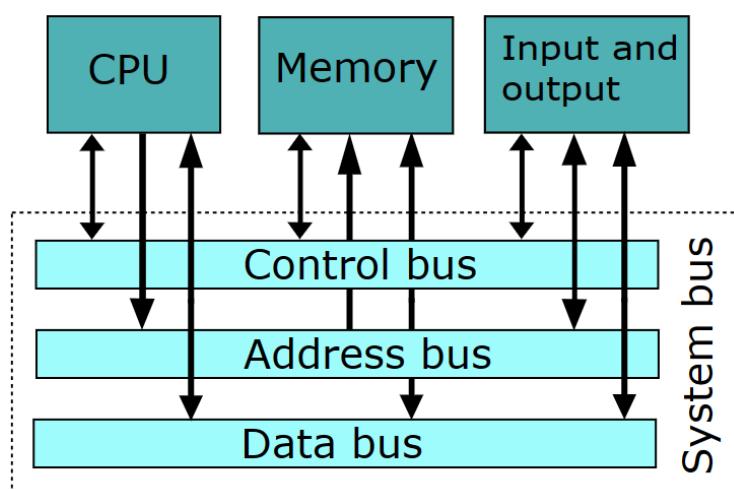
## 입출력장치

입출력장치는 마이크, 스피커, 프린터, 마우스, 키보드 등 컴퓨터 외부에 연결되어 컴퓨터 내부와 정보를 교환하는 장치를 의미한다.

## 메인보드와 시스템 버스

위의 4가지 핵심 부품은 모두 main board에 연결된다. main board에 연결된 부품들은 bus를 통해 서로 정보를 주고받을 수 있다.

컴퓨터의 네 가지 핵심 부품을 연결하는 버스는 **system bus**라고 한다.



### Example

CPU가 instruction을 불러오는 과정

1. control bus로 '메모리 읽기' control signal을 보낸다
2. address bus로 읽을 주소를 보낸다
3. 메모리에서 데이터 버스로 읽은 data를 보낸다

## 02 데이터

### 0과 1로 숫자를 표현하는 방법

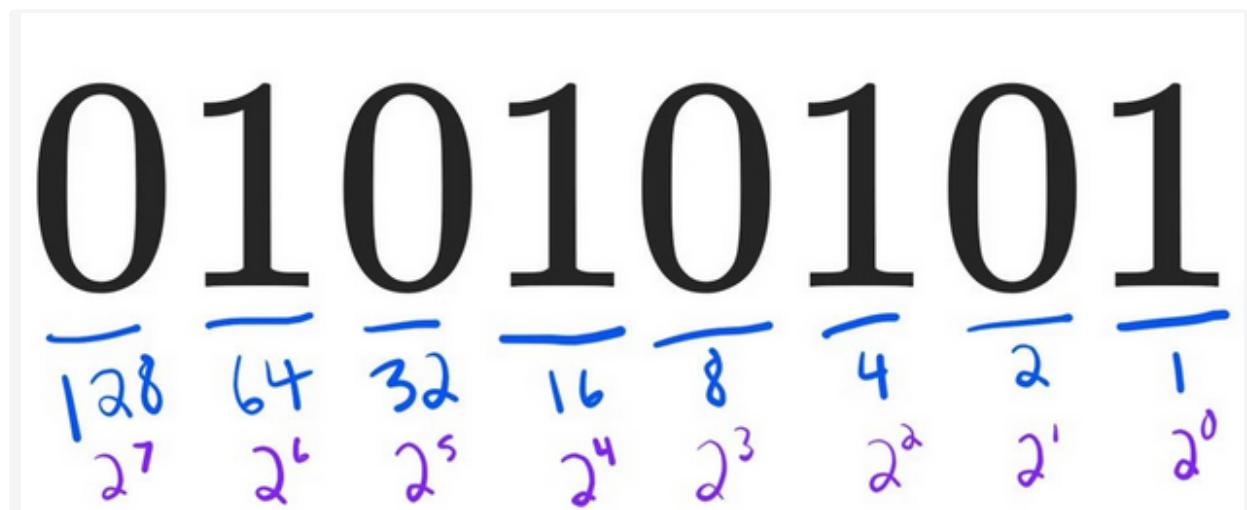
RAM, Shift register 등의 구조가 0과 1만 처리할 수 있게 돼있다.

### 정보 단위

- bit : 컴퓨터는 0과 1만 처리할 수 있으며, 이를 나타내는 가장 작은 정보 단위를 **bit** 라고 한다.
- byte : byte는 8개의 비트를 묶은 단위이며  $2^8$ (256)개의 정보를 표현할 수 있다.
- KB, MB, GB : K는 곱하기  $10^3$ , MB는  $10^6$ , GB는  $10^9$ 이다.
- word : CPU가 한 번에 처리할 수 있는 데이터 크기를 의미한다. 만약 CPU가 한 번에 16비트를 처리할 수 있다면 1워드는 16비트이다.(현대엔 메모리의 한계를 극복하기 위해 64bit가 대세)

### 이진법

0과 1만으로 숫자를 표현하는 방법을 **binary**(이진법) 이라고 한다.



## 이진수의 음수 표현

이진수의 음수는 two's complement를 구해 이 값을 음수로 표현한다.

2의 보수의 사전적 의미는 '어떤 수를 그보다 큰  $2^n$ 에서 뺀 값'이다. 예를 들어 011의 2의 보수는 011보다 큰  $2^3$ , 즉 100에서 011을 뺀 001이 된다.

쉽게 말하면 '모든 0과 1을 뒤집고, 거기에 1을 더한 값'이다.

### two's complement 를 사용하는 이유

컴퓨터의 논리회로는 full adder로 모든 연산을 처리하기 때문이다.(뺄셈기가 없다)

two's complement를 이용하면 덧셈으로 뺄셈을 할 수 있다.

#### Example

2의 보수와 10의 보수는 같은 방식으로 작동하기 때문에 10의 보수로 예를 들면

54 -34

1. 34의 2의 보수=100 -34=66
2. 54 +66=120
3. carry 버리기 → 20

34 -54

1. 54의 10의 보수=100 -54=46
2. 34 +46=80
3. 80의 2의 보수=100-80= 20 (minus 붙이기)

## 16진법

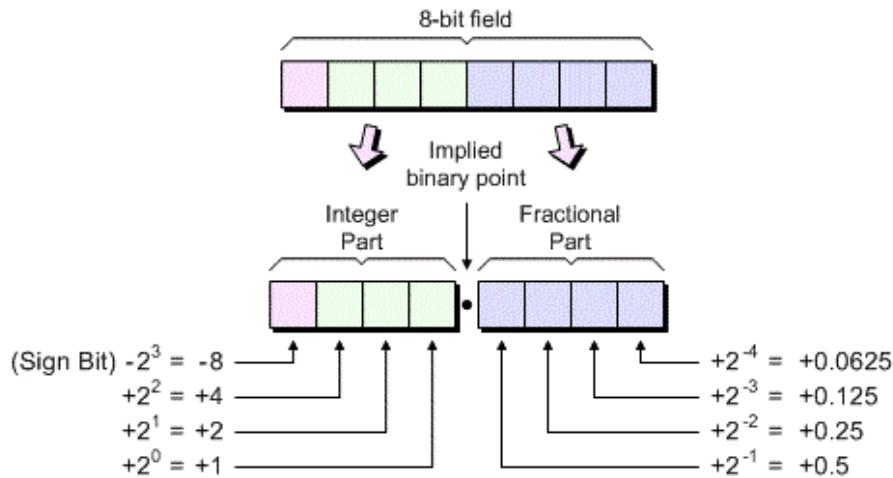
2진수는 길기 때문에 16진수를 이용해 2진수를 줄일 수 있다.

16진수는 한 글자(0~9, A~F)당  $2^4$ 개의 숫자를 표현할 수 있다. 즉, 4bit를 표현할 수 있다.

Example: 1101 0101 → D5

## How to represent decimals

### Fixed point representation



The location for the point to be placed is fixed in advance.

- Advantage : Simple and precise.
- Disadvantage : Too small a range of digits available

### Floating point representation



The floating point is a trade-off between range and precision, which is used the most.

A floating-point number is represented approximately with a fixed number of significand digits and exponent digits.

- Advantage : A wide range of digits are available.
- Disadvantage : There may be an error in the fractional part.

### Example: 5.125 in a floating-point representation

1. Convert 5.125 to binary  
 $5$  (integer part) = 101 (binary)  
 $0.125$  (fractional part) =  $2^{-3} = 1/8 = 0.125 = 001$  (binary)  
 $5.125$  (decimal) = 101.001 (binary)
2. Normalize the binary representation  
 $101.001$  (binary) =  $1.01001 \times 2^2$
3. 32bit Floating-point representation  
**Sign bit (1 bit):** 0 (positive number)

**Exponent (8 bits):** The exponent is determined by adding the exponent part of the normalized representation and the bias (127) ->  $2 + 127 = 129$ , which is represented as 10000001 in binary.

**Significand (23 bits):** The significand is the fractional part -> 01001.

**Result:** 0 | 10000001 | 0100100000000000000000000

5.125 can be represented precisely without any errors. Why 0.1 is not accurate?

It is because 0.1 becomes a repeating fraction 0.0001100110011001100110011..., which is approximated

## 0과 1로 문자를 표현하는 방법

### ASCII code

ASCII는 7bit, 즉 128개의 문자를 표현할 수 있다. (1byte에서 남은 1bit는 오류 검출을 위한 parity bit이다.)



### EUC-KR

0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

b0a0	가	각	간	감	갈	갉	감	갑	갓	갓	강	갓	갓	갓	
b0b0	같	갚	갛	걔	깬	깰	캡	갓	겠	깽	갸	깎	갇	갈	
b0c0	갓	강	개	갠	깰	거	걱	건	걸	깃	검	겁	겄	겅	
b0d0	겅	걸	깃	겋	게	겐	겔	겜	겝	겠	겡	겡	겨	겨	견
b0e0	견	결	겸	결	결	견	결	계	겐	겔	겜	곗	고	곡	곤
b0f0	곤	골	곪	곪	곰	곰	곳	공	곶	꽈	관	꽈	꽝		
b1a0	괌	괌	괌	괌	괌	괌	괌	괌	괌	괌	괌	괌	괌	괌	괌
b1b0	윕	翕	翕	翕	翕	翕	翕	翕	翕	翕	翕	翕	翕	翕	翕
b1c0	굼	굼	굼	굼	굼	굼	굼	굼	굼	굼	굼	굼	굼	굼	굼
b1d0	귈	귈	闺	闺	闺	闺	闺	闺	闺	闺	闺	闺	闺	闺	闺
b1e0	긽	긽	긽	긽	긽	긽	긽	긽	긽	긽	긽	긽	긽	긽	긽

2byte(16진수 4자리)(2350개)로 한글을 표현한다. 2byte는 모든 한글을 표현하기에 부족하기 때문에 CP949같은 방식이 있다.

### Unicode

언어마다 인코딩 방식이 다르면 디코딩이 불편하기 때문에 표준 인코딩 방식이 필요하다.

표준 인코딩 방식으로 **unicode** 가 있으며, unicode에 부여된 값을 인코딩하는 방식으로 UTF-8, UTF-16 등이 있다.

## UTF-8

가장 대중적인 인코딩은 UTF-8이다.

0~007F : 1byte

0080~07FF : 2byte

0800~FFFF : 3byte

10000~10FFFF : 4byte

한글은 3byte로 표현된다.

## 03 명령어

### 소스 코드와 명령어

#### 고급 언어와 저급 언어

##### high-level language

사람이 이해하기 편한 언어

사람이 고급 언어를 이용해 프로그래밍한 후, 저급 언어(instruction)으로 변환되어 실행된다.

##### low-level language

컴퓨터가 이해할 수 있는 언어

- 기계어: 이진수로 돼있는 언어로, 컴퓨터가 직접 실행하는 형태이다. (example: 0101 0101)
- 어셈블리어: 기계어를 1대1 치환하여 사람이 좀 더 읽기 편하게 만든 언어이다. (example: push rbp, mov a, b 등)

#### 컴파일 언어와 인터프리터 언어

고급 언어는 컴파일 혹은 인터프리터 방식으로 저급 언어로 변환된다.

#### 컴파일 언어

컴파일러에 의해 소스 코드 전체가 저급 언어로 변환된다. 소스 코드에 오류가 하나라도 있으면 컴파일에 실패한다.

컴파일러를 통해 저급 언어로 변환된 코드를 **object code**라고 한다.

**장점**

- 일단 컴파일 되면 runtime 속도가 빠르다
- 컴파일러 최적화

**단점**

- 컴파일 시간이 오래걸릴 수 있다

**인터프리터 언어**

소스 코드가 한 줄씩 차례로 저급 언어로 변환되어 실행된다.

소스 코드에 오류가 있어도 그 오류가 있는 line에 가기 전까진 실행될 수 있다.

**장점**

- 컴파일을 기다릴 필요가 없다

**단점**

- runtime 속도가 느리다.

느린 runtime 속도를 해결하기 위해 JIT compile이나 c++로 작성한 코드를 import해서 인터프리터는 interface로 사용하는 방법도 있다.

- optimization).

**Just In time Compile**

JIT compilers typically analyze the code in chunks or blocks to generate machine code just before it is executed. while interpreters execute code line by line.

**목적 파일 vs 실행 파일**

저급 언어로 이루어진 파일을 object file이라고 하며, 실행할 수 있는 파일은 실행 파일(윈도우의 .exe)이라고 한다. object file을 실행 파일로 만들기 위해 linking이라는 작업을 거쳐서 분리된 object file을 합치는 과정이 필요하다.

**명령어의 구조****operation code와 operand****operand**

연산에 사용할 데이터

Example : mov a, b에서 a와 b

## operator

수행할 연산

Example : mov a, b에서 mov. 이 외에도 다양한 명령어가 있다.

즉시 주소 지정 방식

operand field에 데이터를 직접 넣는 방법. 데이터를 찾아서 가져오지 않기 때문에 빠르지만 operand의 크기에 한계가 있다.

직접 주소 지정 방식

operand field에 데이터가 저장된 메모리 주소를 넣는 방법. operand의 크기에 한계가 있기 때문에 주소의 범위에 한계가 있다.

간접 주소 지정 방식

operand field에 데이터가 저장된 메모리 주소를 가리키는 주소를 넣는 방법.

레지스터 주소 지정 방식

직접 주소 지정 방식과 같지만 데이터가 저장된 곳이 레지스터이다.(빠르다)

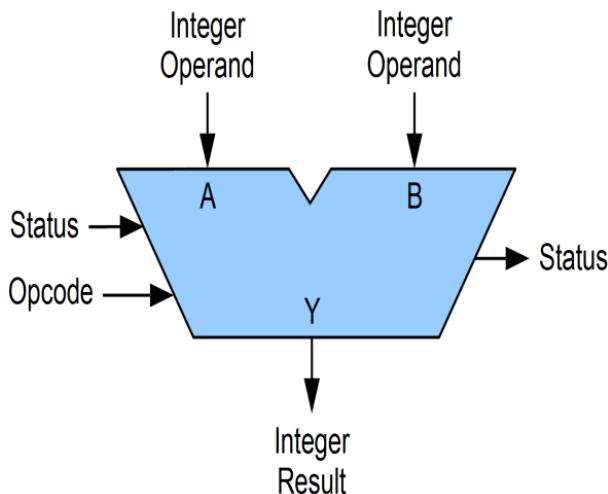
레지스터 간접 주소 지정 방식

간접 주소 지정 방식과 같지만 주소가 저장된 곳이 레지스터이다.

## 04 CPU의 작동 원리

### ALU와 제어장치

#### ALU



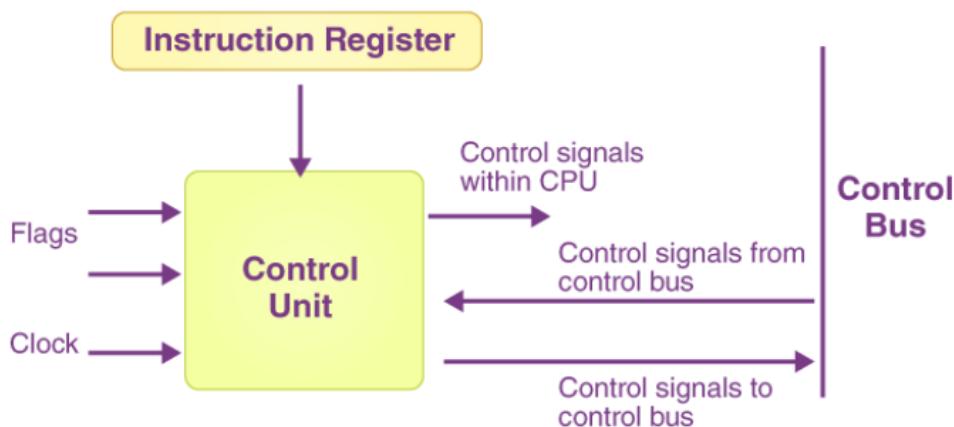
register에게 operand를 받고 Control Unit에게 operator(instruction)를 받아 연산을 수행한다..

레지스터가 메모리보다 훨씬 빠르기 때문에 결과값은 일시적으로 레지스터에 저장된다.

결과값과 더불어 flag를 내보내는데, sign flag, carry flag, overflow flag, interrupt flag 등이 있으며 이는 flag register에 저장된다.

#### 제어장치

제어장치는 명령어를 해석하고 제어 신호를 보낸다



Block Diagram of the Control Unit

### 제어장치가 받는 정보

clock, instruction, flag, control signals from control bus

### 제어장치가 보내는 정보

- CPU내부(ALU에 연산을 지시하기 위해 ALU 내부에 신호, 레지스터에 제어 신호)
- memory, peripherals 등을 제어하기 위해 control bus에 신호

## 레지스터

instruction과 data는 반드시 레지스터에 저장된다.

### Program Counter

memory에서 가져올 instruction의 주소를 저장(instruction pointer라고도 불림)

Program counter의 주소는 읽을때마다 증가해서 다음 명령어를 읽을 준비를 한다.(jump 명령어나 interrupt 발생시에는 아니다)

### Instruction Register

memory에서 가져온 instruction을 저장

### Memory Address Register

접근할 memory의 주소를 저장

### Memory Buffer Register

memory에서 읽은 값이나 memory에 쓸 값을 저장

### General Purpose Register

말그대로 모든지 저장할 수 있는 레지스터이다.

### Flag Register

ALU 연산결과에 따른 flag를 저장

## Stack addressing mode

`stack pointer` : memory의 stack 영역을 가리키는 pointer이다.

## Displacement addressing mode

### Relative addressing mode

program counter에 저장된 instruction의 주소 값에 +3과 같은 상대적인 위치 값을 넣어서 주소를 얻는다.

예를 들어 instruction의 주소가 10000이면 +3을 해서 1003을 얻는다.

### Base-register addressing mode

Base-register에 저장돼있는 주소 값에 +3과 같은 상대적인 위치 값을 넣어서 주소를 얻는다.

## 명령어 사이클과 인터럽트

### 명령어 사이클

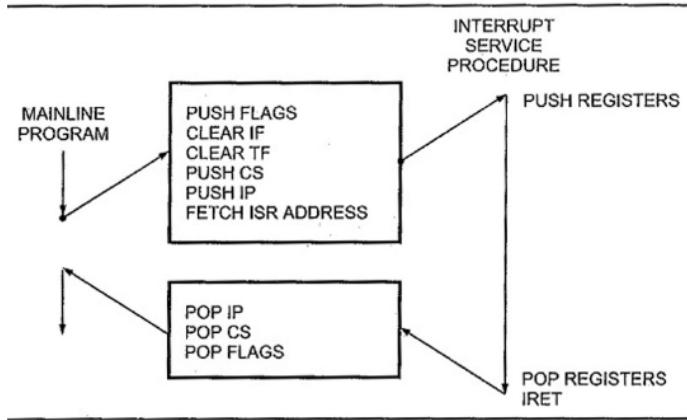
- Fetch cycle: memory에 있는 instruction을 CPU로 가져오는 과정
- Execution cycle: Control unit이 instruction register에 담긴 값을 해석하고 control signal을 발생시키는 과정

### Interrupt

An **interrupt** is a request for the processor to temporarily halt the current task and switch to another task with a higher priority."

- **external interrupt(hardware interrupt)** happens as a result of outside interference such as from the user or from the peripherals. It functions as a notifier.
- **internal interrupt**, also called Trap, happens when wrong instructions or data are used. (exceptions like divided by zero, overflow)
- **software interrupt(system call)**: is a type of interrupt that is triggered by a specific instruction in a program, rather than by an external event or hardware malfunction. It's a mechanism for a program to interrupt the current process flow and request a service from the operating system.  
A program running in user mode needs to send a system call to the operating system to access system resources and perform tasks in kernel mode.

# Interrupt Service Routine



1. An interrupt occurs
2. Current program status is saved onto a stack
3. Jump to the interrupt vector
4. The interrupt is processed
5. Jump to the previous program

## polling VS (vectored) interrupt

- **polling** : Each interrupt is checked by one by one in an infinite loop. This busy-waiting causes overheads.

### Example

```
while True:
    if request==0:
        response0()
    elif request==1:
        response1()
    else
        response2()
```

- **interrupt** : A hardware device notifies that an interrupt has occurred. (more effective) This method has a limit that a hardware support is needed. But it is much faster than polling, so it is essential for real-time response.

## 05 CPU 성능 향상 기법

### 빠른 CPU를 위한 설계 기법

#### multi-core

현대의 CPU는 여러개의 CPU(core)가 모여있는 형태인데, 이를 멀티코어 CPU라고 한다.

clock속도에 한계가 있기 때문에 멀티코어는 성능 향상을 위해 필수적이지만 CPU의 연산 속도는 반드시 core의 숫자에 비례하는 건 아니다.

병렬 처리가 쉽지 않기 때문이다.(직원 숫자 != 일처리 속도)

#### thread

#### hardware thread

2 cores 4 threads는 실제로 동시에 2개의 명령어를 실행할 수 있고, 겉보기엔 4개의 명령어를 동시 처리할 수 있음을 의미한다.(하나의 core가 2개의 명령 처리)

#### software thread

독립적으로 실행되는 명령어 집합 단위이다.

하나의 프로그램은 여러개의 thread를 가질 수 있다.

### 명령어 병렬 처리 기법

#### 명령어 pipeline

##### 명령어 처리 과정

1. Instruction fetch
2. Instruction decode
3. Execute instruction
4. Write back

CPU는 여러 명령어의 다른 단계를 동시에 처리할 수 있다.

그러므로 clock마다 여러 명령어의 다른 단계를 동시에 처리하면, 다음 명령어는 이전 명령어가 끝날 때 까지 기다릴 필요가 없다.



이를 **instruction pipeline**이라고 한다.

## Pipeline hazard

### Data hazard

현재 명령어에 다음 명령어의 data가 필요하지만 다음 명령어의 결과는 아직 알 수 없을 때 발생

### Control hazard

program counter를 따라 다음 명령어를 pipeline에 미리 가져와서 처리 중이었지만 jump로 program counter 값에 예상치 못한 변화가 생겼을 때 발생

### Structural hazard

다른 명령어가 동시에 같은 resource(ALU, register, memory 등)를 사용하려고 할 때 발생

## Superscalar

CPU에 여러 개의 pipeline이 포함된 구조

## Out-of-order execution(async)

In an in-order execution, instructions are processed sequentially the entire pipeline might stall until the hazard is resolved (e.g., waiting for data to be fetched from memory).

Out-of-order execution minimizes these stalls by finding and executing other independent instructions in the meantime.

## Example

1. add a, 1
2. add b, a
3. add c, b
4. 5, 6, 7., and so on

In this case, the third instruction must wait until the first and second are completed.

Out-of-order execution is used to process subsequent instructions that do not depend on the third instruction earlier while waiting for it. This prevents the pipeline from stalling.

## Instruction set

If the instruction set is different, it means the assembly language is different too, which implies that only CPUs with the same instruction set can understand each other's commands.

## CISC (Complex Instruction Set Computer)

In a CISC architecture, instructions can vary in length(variable-length instructions). This flexibility allows for more powerful instructions but can make instruction decoding and pipelining more challenging.

### Advantages

- Since there is a wide variety of instructions, tasks can be performed with a fewer number of instructions.

### Disadvantages

- As the instructions are of variable length, the time to process each instruction is not constant, and executing a complex instruction with multiple operations might require multiple clock cycles. (This can be a critical obstacle for pipelining.)

## RISC (Reduced Instruction Set Computer)

RISC architectures have fixed-length instructions and aim for simplicity and uniformity in their instruction encoding.

### Advantages

- The instructions are standardized, and since each instruction is executed in 1 clock cycle, it is optimized for instruction pipelining.

### Disadvantages

- The number of instructions that make up a program is large (the size of the compiled program is big).

## 06 메모리와 캐시 메모리

### RAM의 특징과 종류

SSD와 같은 non-volatile memory와 다르게 RAM은 volatile memory이다.

non-volatile은 느리지만 용량이 크고, volatile은 빠르지만 용량이 작기 때문에 RAM에는 현재 실행중인 것을 저장한다.

## RAM의 종류

Floating gate transistor

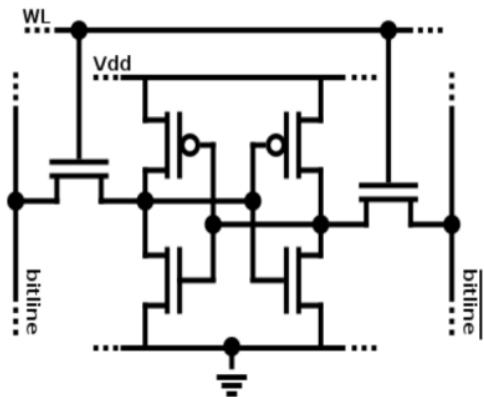


DRAM

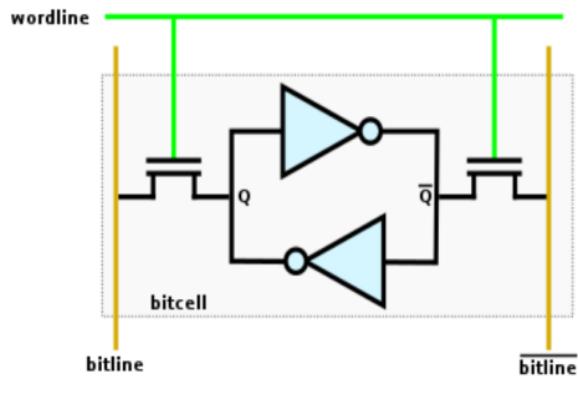


Dynamic RAM의 약자로, Capacitor가 이용되며, 저장된 데이터가 동적으로 변하는(사라지는) RAM이다. Refresh가 필요하고, 이는 DRAM이 volatile인 이유이다.

SRAM



Transistor diagram



Inverter diagram

Static RAM의 약자로, Flip-flop가 이용되 refresh를 하지 않아도 데이터가 유지되는 RAM이다.

	DRAM	SRAM
Refresh	O	X
Speed	Bad	Good
Price	Cheap	Expensive
Density	High	Low

Usage	RAM	Cache
-------	-----	-------

## SDRAM

Synchronous Dynamic RAM은 Clock에 맞춰 동작하는 DRAM을 의미한다.

## DDR SDRAM

Double Data Rate SDRAM은 data rate(대역폭)이 두 배이므로 한 clock당 두 번씩 CPU와 통신할 수 있다.

## 메모리의 주소 공간

- Physical address: 정보가 실제로 저장된 RAM에서의 주소
- Logical address: CPU가 사용하는 0부터 시작되는 주소

## MMU

### Memory Translation

Memory Management Unit는 base register를 이용해 Physical address와 Logical address간의 변환을 수행한다.

예를 들어 base register에 1000이 저장돼있고 logical address는 10이라면, 이를 MMU가 physical address인 1010으로 변환한다.(paging 등이 있기 때문에 실제로 이렇진 않음)

### Memory Protection

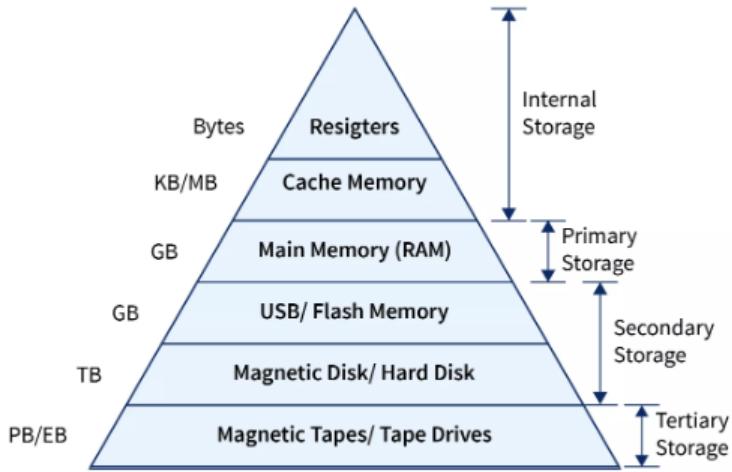
MMU는 limit register를 이용해 어떤 프로그램이 다른 프로그램을 침범하지 못하도록 막는다.

Limit register에 실행 중인 프로그램의 최대 logical address가 저장된다.

CPU는 memory에 접근하기 전에 logical address가 limit register보다 큰지 항상 검사한다. (범위를 넘으면 interrupt(trap) 발생)

## Cache memory

### Memory hierarchy



Memory는 빠를수록 가격대비 용량이 작으며 느릴수록 용량이 크다.

이를 기준으로 memory들을 계층적으로 나타낼 수 있다.

### Cache memory



Register와 DRAM사이에 있는 SRAM기반 memory이다.

DRAM으로는 CPU의 속도를 따라가기 벅차기 때문에 CPU 바로 옆에 SRAM을 두고, 일부 데이터를 이곳에 저장해서 빠르게 쓸 수 있도록 한 것이 Cache memory이다.

Core와 가장 가까운 순으로 L1(Level 1) L2, L3 cache라고 부른다.

### Locality of reference

- Cache hit: CPU가 cache memory에서 원하는 data를 찾았을 때
- Cache miss: 못 찾아서 RAM에서 직접 가져와야 할 때

Cache miss가 자주 발생하면 cache에서 data를 찾는 시간이 낭비되기 때문에 cache가 없는 것보다 안 좋다.

Cache hit ratio를 높이기 위해 Locality of reference 원칙으로 cache에 data가 저장된다.

#### Temporal locality

최근에 접근된 메모리 공간은 곧 또 다시 접근될 확률이 높다.(LRU cache)

예를 들어 2의 배수를 출력하는 프로그램에선 2가 계속 쓰인다.

#### Space locality

접근된 메모리 공간 근처는 곧 접근될 확률이 높다.

메모리엔 관련된 데이터가 모여서 저장되기 때문이다.

예를 들어 구구단을 출력하는 프로그램의 메모리 공간엔 1부터 9까지 모여서 저장돼있다.

## 08 입출력장치

### 장치 컨트롤러와 장치 드라이버

#### 장치 컨트롤러

Device controller는 CPU와 입출력장치 간의 통신을 중개하는 부품이다.

- Data register: Buffering으로 CPU와 입출력장치 간의 전송률 차이를 완화하기 위한 것이며, 데이터가 많으면 대신에 RAM이 사용되기도 한다.
- State register: 입출력 작업의 상태 정보 저장
- Control register: 수행할 입출력 작업에 대한 제어 정보 저장

#### 장치 드라이버



Device driver는 OS와 hardware가 통신할 수 있게 해주는 interface이다. (abstraction)

Device driver가 없으면 hardware의 복잡한 세부사항을 다 알아야 된다.

예를 들어, device driver는 click()이라는 interface를 제공해주고 OS는 마우스의 자세한 작동 방식을 몰라도 click()을 처리할 수 있다.

### 다양한 입출력 방법

#### Programmed I/O

프로그램 속 instruction으로 입출력장치를 제어하는 방법이다.

CPU는 입출력 instruction을 실행하면 device controller에 명령을 보낸다.

#### Interrupt-driven I/O

Programmable Interrupt Controller에 여러 장치 컨트롤러가 연결되어 hardware interrupt 요청들의 우선순위를 판별해 CPU에 알려준다.

## Direct Memory Access

위의 다른 방법들과 달리 CPU를 거치지 않고 입출력장치와 메모리가 직접 통신할 수 있는 방식이다.

입출력장치는 큰 연산이 필요한게 아니기 때문에 CPU를 쓰면 비효율적이다.(CPU bound, I/O bound)

CPU가 입출력 작업을 명령하면(I/O processor가 명령할 수도 있음) DMA가 입출력 작업을 수행하고 CPU에 interrupt를 걸어 완료를 알린다.

### I/O bus

I/O 작업이 system bus를 마비시키지 않도록 Peripheral Component Interconnect(PCI)와 같은 I/O bus를 따로 둔다.

## 09 운영체제 시작하기, 12 Process synchronization

시스템의 자원을 효율적으로 할당하고 시스템 전체를 관리하는 프로그램으로, memory의 kernel space에서 실행된다.

### 운영체제의 큰 그림

#### Dual mode in OS

- User mode: system의 자원에 접근하는 명령어의 직접 실행 불가능
- Kernel mode: 모든 instruction을 실행 가능

User mode로 실행되는 프로그램이 system 자원에 접근하려면 운영체제에 system call을 보내서 kernel mode로 작업을 수행해야 한다. 이는 software interrupt이다.

예를 들어 printf()도 내부적으로 system call을 호출한다.

### 운영체제의 핵심 서비스

- Process 관리
- 자원 접근 및 할당(CPU scheduling, virtual memory, peripherals 등)
- File system 관리 (file의 CRUD, directory 관리 등)

\*\* Hypervisor를 지원하는 CPU는 virtual machine이 kernel mode에서 실행될 수 있도록 해준다.

### Process synchronization

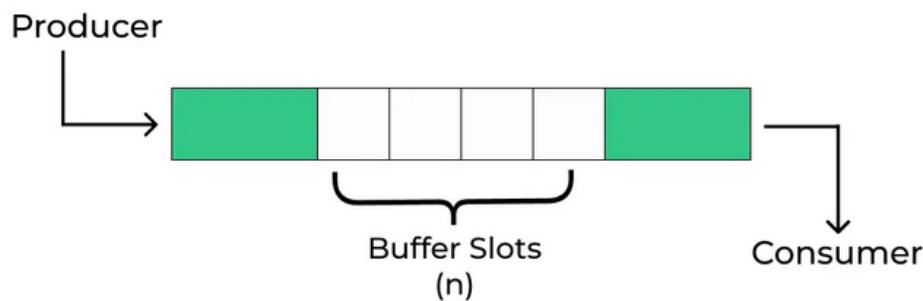
- **Data consistency** refers to whether the same data kept at different places do or do not match.
- **Critical section** is a code segment where shared resources are accessed.
- **Race condition** is a situation where multiple processes are accessing a shared resource(**critical section**) simultaneously, which might affect **data consistency**.

- **Process Synchronization** refers to aligning the execution timing of processes to ensure that shared resources are accessed by only one process at a time to maintain **data consistency**.

## Examples

Producer-consumer problem

## Bounded Buffer Problem



겉보기엔 100개의 data를 넣고 100개의 data를 빼는 작업을 동시에 하면 결국 0개의 data가 될 것 같지만 실제론 아니다.

Process synchronization 없이 producer와 consumer가 buffer에 data를 넣고 빼면 일관적이지 않은 값이 나온다.(실제론 write 작업은 3개의 instruction으로 이루어져 있다)

1. mov r1, var (see page 299)
2. mov r1, "text"
3. mov var (see page 299), r1

Readers, writer problem

읽기는 데이터를 바꾸지 않고 data consistency를 유지하기 때문에 reader 프로세스들은 공유 자원에 동시에 접근할 수 있다. 하지만 쓰기 도중엔 읽거나 쓰면 data consistency가 깨지기 때문에 writing이 끝날 때 까지 기다려야 한다.

## 동기화 기법

### Lock의 종류

- **Mutex(mutual exclusion)** : Used to ensure that a shared resource is accessed only by one thread at a time. It has two states: locked and unlocked.

- **Semaphore** : A mutex is a binary semaphore. A semaphore can allow more than one thread to access a shared resource but limits the number.(can be used as a counter)
- **Monitor**: Unlike mutexes, where a thread must explicitly lock and unlock, monitors manage locks implicitly. When a thread enters a monitor's method, it automatically acquires the lock, and when it exits the method (either normally or by waiting on a condition variable), it automatically releases the lock. Monitors can use a condition variables, which are used to block a thread until a particular condition is met.

## Lock의 구현

- **Spinlock(busy waiting)**: When a thread attempts to acquire a spinlock that is already held by another thread, it will continuously check (or "spin") until the lock becomes available. This approach can be efficient if the expected wait time is short since the thread remains active and does not enter a sleep state. However, it can also waste CPU cycles if the lock is held for a long time.
- **Process queue**: The PCB of a process is put in a queue and popped when the shared resource has become available.

### Semaphore 코드

```

class Semaphore{
    int shared_resource; // 1 or more (If this is initialized with the value 1, it
    can function similarly to a mutex.)
    Queue process_queue;
}

void acquire(Semaphore* S, Process* process){ // When a process wants to take a
shared resource
    S->shared_resource--;
    if ( S->shared_resource < 0 ) { // Put the process to sleep if other processes
are using the shared resource
        S->process_queue.enqueue(process);
        sleep(process);
    }
}

void release(Semaphore* S) { // Wakes up a process that wanted to take a shared
resource
    S->shared_resource++;
    if ( S->shared_resource <= 0 ) {
        process=S->process_queue.dequeue();
        wakeup(process);
    }
}

```

### Mutex 사용 예시

**mutex**

```

from multiprocessing import Process, Value, Lock

def add_100(number, lock):
    for i in range(100): # with lock: <- This "monitor" can be used instead of
specifying lock.acquire() and release() manually.
    lock.acquire()
    number.value += 1 # Critical section
    lock.release()

def add_100_monitor(number, lock):
    for _ in range(100):
        with lock: # Monitor used. This will acquire the lock and automatically
release it after the block
            number.value += 1 # Critical section

if __name__ == "__main__":
    semaphore = Lock() # Semaphore
    shared_number = Value('i', 0) # i = integer
    print("Start from", shared_number.value)
    p1 = Process(target=add_100, args=(shared_number, semaphore))
    p2 = Process(target=add_100, args=(shared_number, semaphore))
    p1.start()
    p2.start()
    p1.join()
    p2.join()
    print("End with ", shared_number.value)

```

**13 deadlock****Deadlock**

Deadlock is a situation where processes are waiting for each other to release their shared resource and none of them ever does.

Improper synchronization results in deadlock.

**4 conditions of deadlock**

1. **Mutual Exclusion:** Only one process can hold a resource at a time
2. **Hold and wait:** Holding a resource while waiting
3. **No preemption:** A held resource cannot be forcefully taken away

4. **Circular wait:** Processes are waiting for resources held by each other.

### **Deadlock prevention:**

Deadlock can be avoided by getting rid of one of the 4 conditions of deadlock.

1. **Mutual Exclusion:** Allow multiple processes to access shared resources simultaneously.
2. **Hold and wait:** Require processes to acquire all necessary resources before execution, eliminating the possibility of holding a resource while waiting for others.
3. **No preemption:** Allows resources to be forcibly taken away from processes.
4. **Circular wait:** Assign a unique numerical identifier to each resource and require processes to request resources in ascending order of these identifiers.

Example of circular wait solution:

Let's say resource 2 has a higher priority.

1. Process 1, holding resource 1, realizes it needs resource 2 to proceed.
2. Since resource 2 has a higher identifier than resource 1, Process 1 decides to release resource 1.
3. Process 2, holding resource 2, can now acquire resource 1
4. Process 2 completes its task and releases both resource 1 and resource 2.
5. Process 1 can now acquire resource 2 and continue its execution.

### **Banker's algorithm**

Banker's algorithm ensures that resource requests will not lead to deadlock by checking whether the system will be in a safe state before granting resource allocation.

It checks if resources can be allocated without causing a resource shortage to prevent the deadlock.

### Example



$$\text{Max} = \text{Need} + \text{Allocation}$$

1. P0 has been allocated A:0, B:1, C:0 and needs A:7, B:4, C:3 to complete its task.
2. However, there are only A:3, B:3, C:2 available, so P0 cannot be completed at the moment.
3. P1 can be completed because it needs A:1, B:2, C:2, which can be satisfied using the available resources.
4. P1 is completed and its allocated resources (A:2, B:0, C:0) are released and added to the available resources, which becomes A:5, B:3, C:2.

## Dining philosopher problem



It can be described with either forks or chopsticks. The problem was designed to illustrate the challenges of avoiding deadlock.

*The dining philosophers problem states that there are 5 philosophers sharing a circular table and they eat and think alternatively.*

*A hungry philosopher may only eat if there are both chopsticks available. Otherwise a philosopher puts down their chopstick and begins thinking again.*

*The dining philosophers is a classic synchronization problem as it demonstrates a large class of concurrency control problems.*

### How philosophers behave

1. think until the left chopstick is available; when it is, pick it up;
2. think until the right chopstick is available; when it is, pick it up;
3. He picks up another chopstick too
4. when both forks are held, eat for a fixed amount of time;
5. put the left fork down;
6. put the right fork down;
7. repeat from the beginning.

This satisfies all of the 4 conditions of deadlock

### Solution of Dining Philosophers Problem

1. **Mutual Exclusion:** Cannot be allowed because it is a basic condition of the problem.
2. **Hold and wait:** A philosopher must acquire both chopsticks simultaneously.
3. **No preemption:** Cannot be allowed because this is a basic condition of the problem.
4. **Circular wait:** The "resource hierarchy" approach can be used. Assign a unique identifier to each and require the philosophers to always acquire the chopsticks in a defined order.

## 14 가상 메모리

### Contiguous memory allocation

#### First fit

The partition is allocated to the first sufficiently large space from the top of memory. The OS searches for the first available partition with a sufficient size.

- It is fast in processing but can waste a lot of memory.



#### Best fit



The entire list of free partitions is searched through to find the smallest hole that is adequately large.

- By allocating the smallest adequate hole, it minimizes wasted space in memory. This makes it an effective method for saving memory and reducing fragmentation, particularly in scenarios where numerous small allocations are made.
- Slower than First fit.



The entire list of free partitions is searched and the biggest hole possible is allocated.

#### External fragmentation



Contiguous memory allocation may result in External fragmentation because deallocated a process may leave its memory space unavailable for bigger processes.

This can be resolved by rearranging the processes so that there are no empty spaces, which is known as memory compaction.

However, moving the information loaded in memory can result in significant overhead.

#### Swapping in contiguous memory allocation

Swapping refers to the process of temporarily moving processes that are not currently being executed out of the memory to a secondary storage device. This technique is used to free up space in the limited memory. By using swapping, the effectively available memory space can be increased.

- Swap Space: This is the area on a secondary storage device where processes are temporarily stored. Swap space acts as an extension of the system's physical memory, allowing for more processes to be loaded than would otherwise fit into the physical RAM.
- Swap-Out: This process involves moving a process from the main memory to the swap space. Swap-out is typically performed when the system needs to free up RAM for other processes or when the system determines that a process is idle or less critical.
- Swap-In: Conversely, swap-in is the process of moving a process back from the swap space into the main memory. This occurs when the process is needed for execution. The system will then allocate space in the physical memory for the process and reload it from the swap space.

## Non-contiguous memory allocation

Non-contiguous memory allocation is a method of storing data in multiple memory locations rather than in a single contiguous block.

## Techniques

- Paging(majority): Divides a process's memory space into fixed-size blocks called pages. Each process is divided into pages of the same size, which can be stored anywhere in the physical memory.
- Segmentation(unusual): Divides a process's memory space into different segments, where each segment represents a specific type of data or code (e.g., code segment, data segment, stack segment). They can be non-contiguously allocated.

## Challenges

- More complex memory management.
- Overhead of maintaining page tables or segment tables.

## Virtual memory

Virtual memory is a memory management technique that creates an illusion of a very large memory space by using both physical memory (RAM) and a part of secondary storage (like a hard disk).

Virtual memory inherently relies on non-contiguous memory allocation.

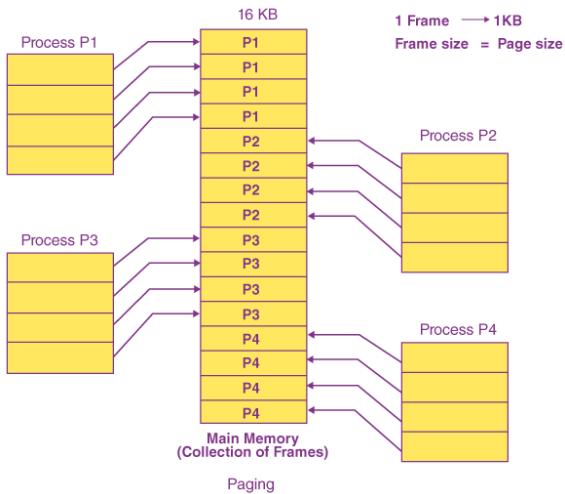
## Benefits

- Provides a large address space to processes, irrespective of the actual physical memory available.

## Considerations

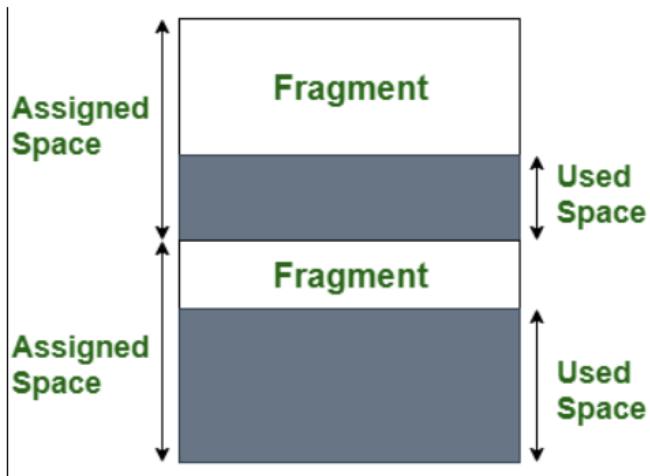
- Overhead of page table management.
- Performance can be affected by the paging process, especially if the system frequently accesses data that is not in the main memory (page faults).

## Paging



- External fragmentation occurs when processes of various sizes are allocated memory space contiguously.  
This issue can be resolved by dividing processes into uniform units called "pages" and slicing the memory space into equally sized units known as "frames", which are the same size as pages. Then, pages are allocated to frames.
- Paging is typically used in conjunction with virtual memory.

## Internal fragmentation



When a piece of allocated memory leaves some unused space.

For example, when a piece of memory whose size is 10 bytes has 8 bytes of data, leaving 2 bytes unused

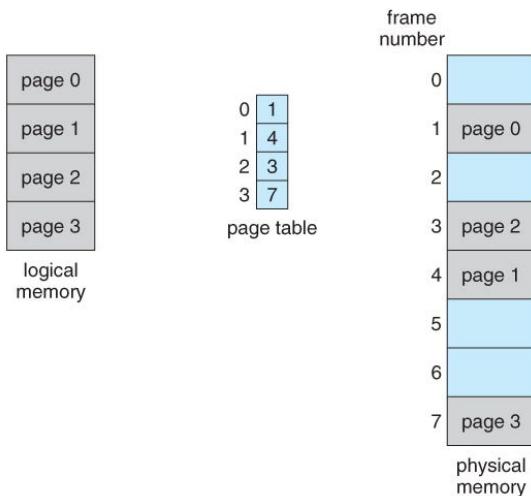
## Page swapping

Page swapping refers to a scenario in which a page from the main memory is replaced by a page from the secondary memory. Page replacement occurs due to page faults.

A **page fault** occurs when a requested page is currently not loaded into the main physical memory.

- **Page-out:** Moving a page from the main memory to the secondary memory.
- **Page-in:** If the data is not in RAM, a page fault occurs, and the OS loads the required data from the secondary storage to the main memory.

## Page table



The OS maintains a page table to track which virtual pages are mapped to which physical frames of memory. The virtual memory address is translated to a physical address using the page table.(page number + offset)

For fast page access, the page table is stored in a cache called **TLB**.(Translation Lookaside Buffer)

## Page Table Entry

Each row in a page table is referred to as a page table entry, which stores data beyond just the page number and frame number. This includes:

- **Valid Bit**: Indicates whether the corresponding page is accessible. Essentially, it informs if a page fault has occurred, necessitating a page swap.
- **Protection Bit**: Specifies whether the page is read-only or if write operations are also permitted.
- **Reference Bit**: Shows whether the page has been accessed since it was loaded into RAM.
- **Modified Bit**: Indicates whether data has been written to the page, informing about any modifications.

When a page is swapped in, the corresponding page on the secondary storage does not disappear. Therefore, if the data has not been modified after being swapped in, there is no need to write it back to the secondary storage during a swap-out, saving resources. The modified bit is used for this purpose.

## Page replacement

필요한 page만을 RAM에 적재하는 방법을 demand paging이라고 하는데, demand paging을 하다보면 언젠가 RAM이 가득 차게 된다.

공간을 확보하기 위해 page-out이 필요한데, 이를 Page replacement라고 한다.

## Page-replacement algorithms

The fewer the number of page faults, the better the algorithm.

- **First-In First-Out** : The oldest page is moved out. This is the simplest but might be bad. Because even if a page is used throughout the program, it can be paged-out just because it was loaded first.  
Second chance page replacement : To solve the problem of FIFO, there is a method where, before a page-out, the reference bit is checked, and if the page has been accessed, it is moved to the position of a recently loaded page
- **Optimal page replacement** : The page that will be used the least in the future is the one that gets moved out. Theoretically, it guarantees the lowest rate of page faults, but predicting such pages is difficult, making it impractical.
- **Least-Recently-Used** : The page that was used the least recently gets paged out as it is likely not to be used constantly. In other words, the page that has not been used for the longest time is replaced. This is one of the practically best algorithms.

## Thrashing and frame allocation

### Thrashing

In fact, the most significant cause of increased page fault frequency is often not the page replacement algorithm, but rather the small number of frames allocated to a process.

If there were an infinite number of frames, page replacement would not be necessary at all.

While it may seem that increasing parallel processing can enhance speed, it inevitably reduces the number of frames available for each process, leading to frequent page faults and significant time spent on page replacement.

This phenomenon is known as **thrashing**, and to prevent it, frames must be allocated appropriately.

### Frame Allocation Methods

- Equal Allocation: This method involves providing an equal number of frames to each process. It is inefficient because the size of each process can vary significantly, leading to underutilization or overutilization of memory resources for individual processes.
- Proportional Allocation: Frames are allocated in proportion to the size of the processes. One drawback of this method is that larger processes might end up with fewer frames than they actually need.

- **Page-Fault Frequency:** This approach allocates more frames to processes with higher page fault frequencies. It operates under the assumption that a high rate of page faults indicates a need for more frames to reduce these faults.
- **Working Set Model:** Frames are allocated based on the size of the process's working set, which is the set of pages the process has referenced over a specific recent period. This method tries to tailor frame allocation to the actual working memory needs of each process.