

Introduction

minimum embedded system functionality commercial off the shelf processor
or microcontroller no input / output functionality
(minimization or license)

CPU - I/O interface

① parallel → N independent lines connect CPU with I/O and one word or

block of word in transfer

using operation

② serial → M ($M \ll N$) lines connect CPU with I/O and 1 bit in transfer

using operation over serial protocol (SPI, I2C, USB)

- CPU uses I/O multiplexing asynchronous (CPU in software, I/O instruction)
- Read / Write operation in init for CPU
- method initiates CPU in data ready to read

1.) Polling → now check availability periodically

Advantage

- Simple to implement

Disadvantages

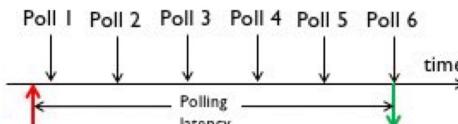
- High latency as I/O are polled serially (time when the I/O is served – time when the peripheral needs service)
- CPU time is wasted when polled devices are not to be serviced.

- 10 devices; only device 6 needs service.

Example

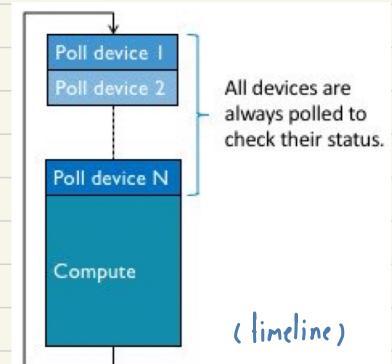
- Device A provides 1 byte.
- Device B provides 2 bytes.

```
while(1)
{
    if(A_is_ready())
    {
        resA = read_byte_from_A();
    }
    if(B_is_ready())
    {
        resB_1 = read_byte_from_B();
        resB_2 = read_byte_from_B();
    }
} Device A provides 1 byte.
Device B provides 2 bytes.
```



Device 6 needs service.
(latency)

Device 6 is serviced.



Device 6 is serviced.

2.) Interrupt → peripheral通知CPU CPU is data ready to read

(CPU가 데이터를 읽을 준비되었을 때 CPU는 인터럽트 신호를 받고 인터럽트 신호를 처리하는 코드를 실행합니다.)

Advantage

- Low latency (time when the peripheral is served - time when the peripheral needs service)

Disadvantage

- Higher hardware complexity
- 10 devices; only device 6 needs service.

If the CPU is running with interrupts enabled, the latency is equal to one instruction.

Device 6 needs service.
Device 6 is serviced.

time

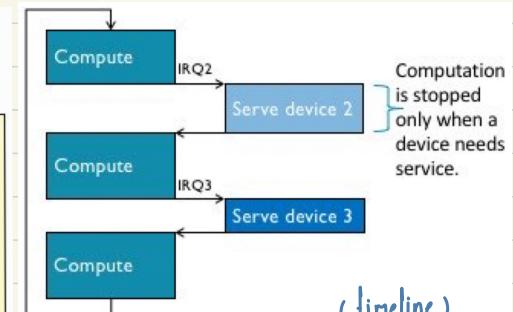
(latency)

Example

- Device A provides 1 byte (IRQ1)
- Device B provides 2 bytes (IRQ2)

```
IRQ1()
{
    resA = read_byte_from_A();
}

IRQ2()
{
    resB_1 = read_byte_from_B();
    resB_2 = read_byte_from_B();
}
```



- data transfer 시스템은 CPU를 통해 데이터를 메모리에 옮기거나 I/O 장치에 전송합니다.

I/O: DMA (which releases CPU from data transfer)

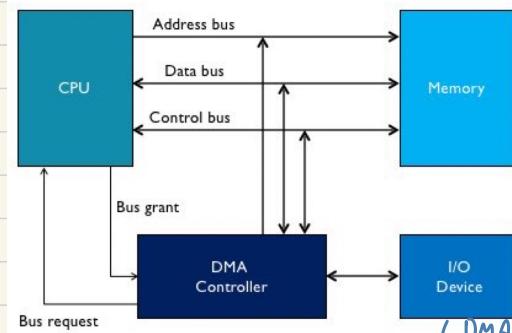
character based transfer → 단위로 데이터 전송 (8/16/32.. bits)

block based transfer → 단위로 데이터 클러스터 전송 (64/128.. bytes)

Direct Memory Access (DMA) Transfer Modes

- Burst - data는 block 단위로 전송되는 방식입니다.

- CPU는 데이터가 전송되는 동안 비활성화됩니다.



(DMA Architecture)

- Cycle stealing - DMA는 1byte 단위로 Bus를 차지하는 방식입니다.

- 1byte 단위로 데이터를 전송하는 동안 CPU는 다른 작업을 수행합니다.

- Transparent - DMA는 CPU가 다른 작업을 수행하는 동안 데이터를 전송하는 방식입니다.

I/O Taxonomy

Output devices which actuate commands via:

- Analog signals: output pins carrying voltages or currents
- Digital level-triggered discrete signals: output pins forming parallel bus carrying digital voltage levels
- Digital pulse-width modulated (PWM) discrete signals: output pins forming parallel bus carrying digital square waveforms with given frequencies and duty cycles
- Bus-based signals: output pins implementing serial communication protocols

Inputs devices which acquire sensors status via:

- Analog inputs: input pins carrying voltages, which required A/D conversion
- Digital level-triggered discrete signals: input pins carrying digital voltage levels
- Digital pulse-width modulated discrete signals: input pins forming parallel bus carrying digital information in the form of pulses duration and/or number of pulses
- Bus-based signals: input pins implementing serial communication protocols

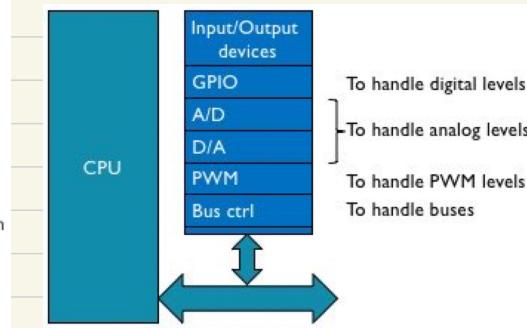
Typical Operations

Output devices

- Generate analog levels through D/A
- General digital levels
- Generate PWM signals
- General bus transfer

Input devices

- Acquire an analog value though A/D conversion
- Read digital level
- Measure timing/repetition of digital pulses
- Read bus transfer



Linux Devices

• Linux provide an abstraction $\text{Linux} \rightarrow \text{Devices}$ I/O

• Linux recognize 3 classes of devices

1.) character devices \rightarrow stream of words

2.) block devices \rightarrow multiples of one block

3.) network interfaces \rightarrow in charge of sending/receiving data over network

Virtual File System (VFS) Abstraction

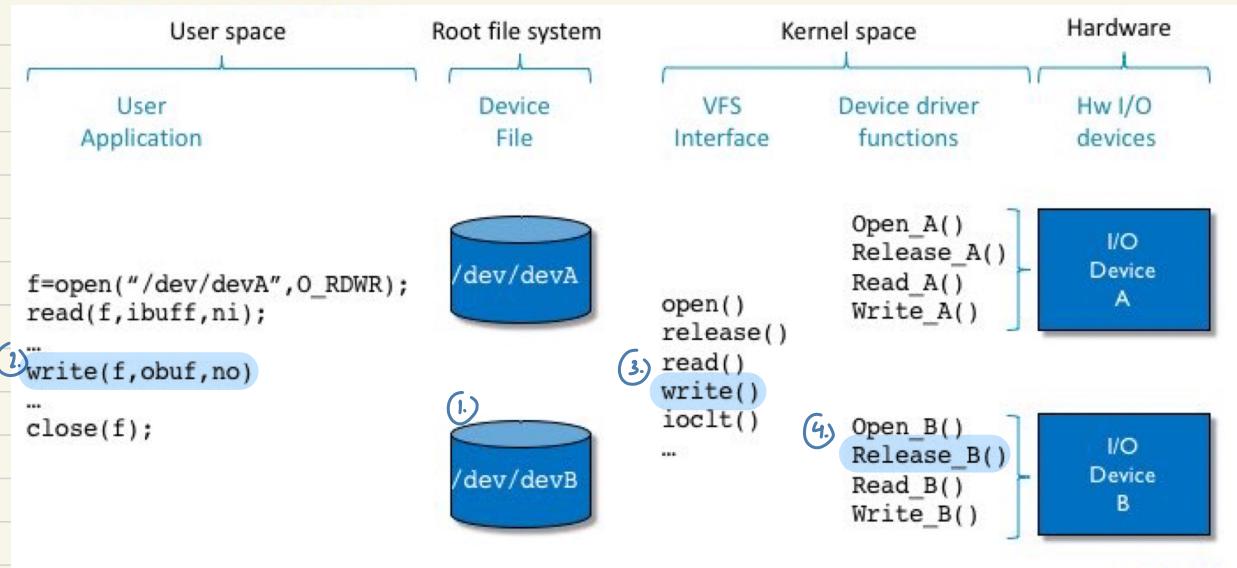
• character / block device \rightarrow file system

Linux forwards the open/read/write/close operations to the I/O device associated to the device file.

The operations for each I/O device are implemented by a custom piece of software in the Linux kernel: the **device driver**.

Typical usage:

- Open the device file
- Read/Write data from/to device file
- Close the device file



- (1) The root file system shall host one device file for each I/O device the user application needs to use.
- (2) The user application deals with the I/O device using the file abstraction: data are read/written to the device file associated with the I/O device.
As with regular files, the device file shall be opened before use and closed after use.
The low-level I/O primitives are used as defined in `fcntl.h/unistd.h`.

- (3) The VFS establishes the association between the low-level I/O primitives used in the user application and the corresponding device driver functions. For example:

User application	VFS	Device Driver
<code>f=open("/dev/devA", O_RDWR);</code>	\rightarrow	<code>open() → Open_A()</code>

(request to open) (open core) (open specific device)

- (4) Implements I/O device-specific operations

(commonly VFS function)

- `ssize_t (*read) (struct file *, char * __user, size_t, loff_t *)`: it reads data from a file.
- `ssize_t (*write) (struct file *, const char * __user, size_t, loff_t *)`: it writes data to a file.
- `int (*ioctl) (struct inode, struct file *, unsigned int, unsigned long)`: it performs custom operations to the file.
- `int (*open) (struct *inode, struct file *)`: it prepares a file for use.
- `int (*release) (struct inode *, struct file *)`: it indicates the file is no longer in use.

Device File Concept

- ឯកសារនេះជាការប្រព័ន្ធនៃការប្រើប្រាស់ឯកសារនៅក្នុងឯកសារទូទៅរបស់ប្រព័ន្ធ
- device file នេះគឺ contain សម្រាប់ផ្តល់ព័ត៌មានទូទៅទៅការប្រើប្រាស់ឯកសារទូទៅរបស់ប្រព័ន្ធ

- The **device file type**, which could be either **c** = character device, **b** = block device, or **p** = named pipe (inter-process communication mechanism)
- The **major number**, which is an integer number that identifies univocally a device driver in the Linux kernel
- The **minor number**, which is used to discriminate among multiple instances of I/O devices handled by the same device driver

Linux Kernel Modules

- device drivers provide an I/O device abstraction via VFS abstraction and are located in kernel space
- device drivers are linked into Linux kernel and are executed in system bootstrap
- device drivers are kernel module files that are loaded in runtime with suitable system program when Linux kernel is booting

System programs

- mknod, to create a device file
- insmod, to insert the module in the kernel
- rmmmod, to remove the module from the kernel
- lsmod, to list the modules loaded in the kernel

Functions provided by a kernel module:

- Initialization function, called upon the execution of the insmod system program, takes care of making Linux aware that a new device driver is available
- Clean-up function, called upon the execution of rmmod, to remove the device driver from the Linux Kernel
- Custom-specific implementations of the VFS abstraction

Linux Kernel Modules

Initialization Function

Data structure containing the major number and the first minor number for the module. It identifies univocally the module in the kernel. It shall be used when creating the device file associated with the module.

```
static dev_t dummy_dev;
```

Data structure used to describe the properties of a character device

```
struct cdev dummy_cdev;
```

Buffer used for displaying output messages on the Linux console

```
{ struct file_operations dummy_fops = {  
    .owner = THIS_MODULE,  
    .read = dummy_read,  
}; }
```

Data structure used to associate the VFS functions to their module-specific implementations. In this example, the read() VFS function is implemented by the dummy_read() function.

```
static int __init dummy_module_init(void)
```

```
{
```

```
    printk(KERN_INFO "Loading dummy_module\n"); : printf()
```

```
① alloc_chrdev_region(&dummy_dev, 0, 1, "dummy_dev");
```

```
② printk(KERN_INFO "%s\n", format_dev_t(buffer, dummy_dev));
```

```
③ cdev_init(&dummy_cdev, &dummy_fops);
```

```
④ dummy_cdev.owner = THIS_MODULE;
```

```
⑤ cdev_add(&dummy_cdev, dummy_dev, 1);  
    return 0;  
}
```

The module initialization function. It is executed as soon as the module enters the Linux kernel and takes care of making the Kernel aware that the new module is available.

It registers a range of character device numbers with the function.

(1.)

```
int alloc_chrdev_region(dev_t *dev, unsigned baseminor, unsigned count, const char *name)
```

where:

`dev` is the dynamically-selected major number of the module;
`baseminor` is the first minor number for the module;
`count` is the number of minor number to reserve for the module;
and `name` is the module name.

(2.)

It prints on the Linux console the major number and the minor number associated with the just-registered character device.

(3.)

It initializes the character device data structure.

(link with X)

```
void cdev_init( struct cdev *cdev, const struct file_operations * fops);
```

where:

`cdev` is the structure to initialize and
`fops` is the `file_operations` for the device.

(4.)

It sets the owner of the module using the `THIS_MODULE` macro.

(5.)

It adds the character device to the Linux Kernel

(link with X)

where:

`p` is the character device structure already initialized;
`dev` is the major number for the device;
and `count` is number of minor numbers for which the device is responsible.

Clean-up Function

```
static void __exit dummy_module_cleanup(void)
{
    printk(KERN_INFO "Cleaning-up dummy_dev.\n");
    cdev_del(&dummy_cdev); → It removes the character device from the
    unregister_chrdev_region(dummy_dev, 1); } Linux kernel.
```

)

It frees the range of major/minor numbers previously registered.

Custom VFS Function

Implementation of the VFS function to read from a file, where `filp` is the pointer to the data structure describing the opened file; `buf` is the buffer to fill with the data read from the file; `count` is the size of the buffer, and `f_pos` is the current reading position in the file.

```
ssize_t dummy_read(struct file *filp, char __user *buf, size_t count, loff_t *f_pos)
{
    printk(KERN_INFO "Dummy read (count=%d, offset=%d)\n", (int)count, (int)*f_pos );
    return 1;
}
```

It prints a message to show that the read function has been executed.

It returns the number of bytes read from the file.
Returning 0 will block the caller application, which will wait until at least one byte is returned.

Quiz

1. "What components are required for interfacing with custom IO devices?"

- A root file system and a device driver
- A kernel space application and a device driver
- A user space application and a device driver
- An inode and a device driver

2. "Which of the following are methods that the CPU can use to recognize that an IO has data ready to be read?"

- Interrupt
- Request
- Observe
- Polling

3. "Which of the following are transfer modes of 'direct memory access'?"

- Synchronous
- Burst
- Transparent
- Cycle stealing

4. "Which of the following is not a Linux recognized device class?"

- Character devices
- Block devices
- Input devices
- Network interfaces

5. "What is a device file?"

- An intermediary through which a kernel application can exchange data with a device driver
- An intermediary through which a user application can exchange data with a device driver
- An intermediary through which a user application can exchange data with the kernel
- A file that hosts the information regarding a specific device

LAB

• Linux add file only in mod vs code

in mod gedit or vim will have spacing now run bitbake