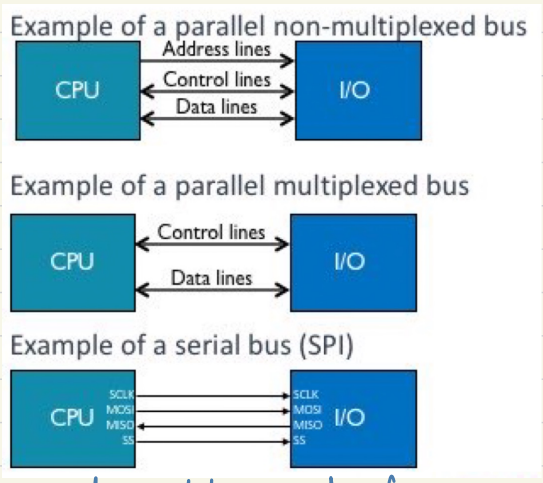
Introduction

การพัฒนาของ embedded system โดยทั่วไปจะต้องเชื่อมต่อ commercial off the shelf processor หรือ microcontroller กับ input / output ที่กำหนดเองได้ (มากที่มีราคาตามท้องตลาด หรือ มี license)

CPU - I/O interface



1. parallel → N independent lines connect CPU with I/O and one word (8/16/32 bits) or block of word จะถูก transfer ใน 1 แต่ละ operation

2. serial → M (M << N) lines connect CPU with I/O and 1 bit จะถูก transfer ใน 1 แต่ละ operation ตาม serial protocol (spi, I2C, USB)

- CPU และ I/O ทำงานกันแบบ asynchronous (CPU ใช้ software, I/O กำหนดตัวเองได้)
- Read / Write operation จะถูก init โดย CPU
- method ที่ทำให้ CPU รู้ว่า data ready to read

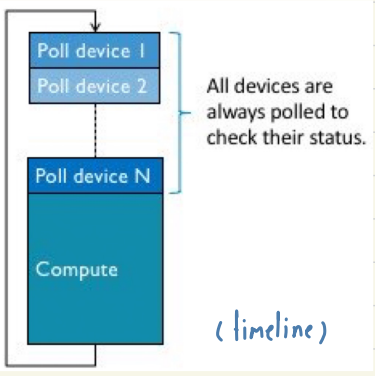
1.) Polling → คอย check ตลอดเวลาแบบ periodically

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.

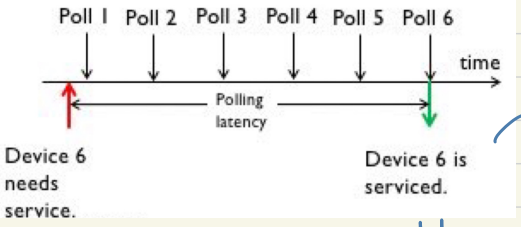
Example
- Device A provides 1 byte.
- Device B provides 2 bytes.

```
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.



(timeline)

- 10 devices; only device 6 needs service.



จะเห็นว่าทางอาจได้รับ service ไม่ทัน

(latency)

2.) Interrupt → peripheral จะเป็นคนบอก CPU ว่า data ready to read

(ตอนจบของแต่ละคำสั่ง CPU จะ check for IRQ line (interrupt request)

Advantage

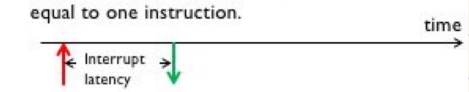
- Low latency (time when the peripheral is served – time when the peripheral needs service)

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.



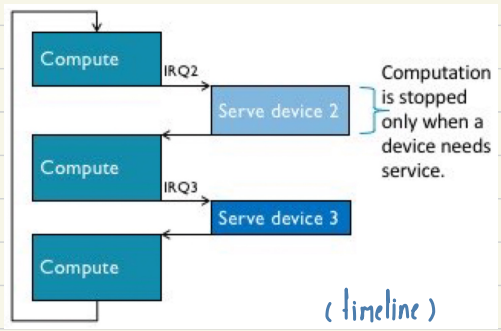
(latency)

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();
}
```



(timeline)

- data transfer รวมๆก perform โดย CPU (move data จาก memory ไป I/O ผ่าน program)

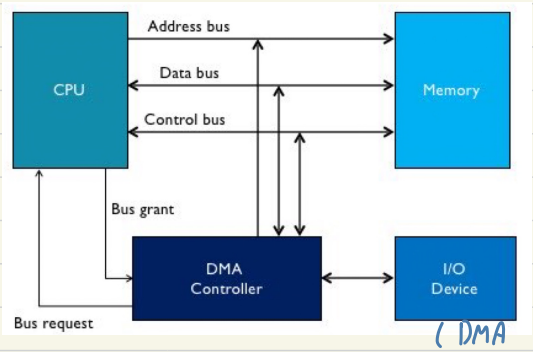
  แบ่ง: DMA (which releases CPU from data transfer)

  - character based transfer → ส่งข้อมูลแบบทีละ: word (8/16/32.. bits)
  - block based transfer → ส่งข้อมูลแบบ clusters of bytes (64/128.. bytes)

Direct Memory Access (DMA) Transfer Modes

1.) Burst - data ทั้ง block จะถูกส่งไปในทีเดียว แบบต่อเนื่องกัน

- CPU จะมีช่วงเวลาที่เป็น inactive นาน (จนกว่าจะส่งข้อมูลเสร็จ)



(DMA Architecture)

2.) Cycle stealing - DMA จะ: transfer data 1 byte แล้วจากนั้น Bus ก็จะว่าง ให้เป็นการ control ของ CPU

- มี issue request ซ้ำๆต่อเนื่อง transfer 1 byte ของ data ต่อ 1 request
- ใช้เวลานาน

3.) Transparent - DMA transfer เอง ในขณะที่ CPU performing operation โดยไม่ใช้ system buses

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

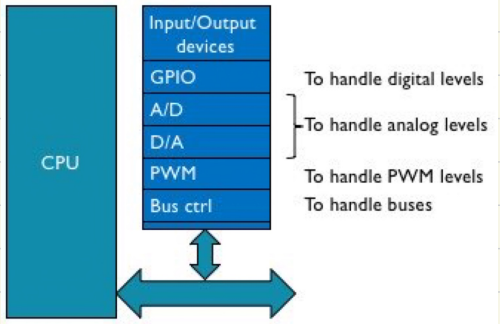
Typical Operations

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 ในการสื่อสารกับ I/O
- Linux recognize 3 classes of devices
  1.) character devices → สามารถถูกเข้าถึงได้แบบ stream of words
  2.) block devices → สามารถเข้าถึงได้แบบ multiples of one block เท่านั้น
  3.) network interfaces → in charge of รับ/ส่ง ข้อมูล เช่น network subsystem ของ kernel

Virtual File System (VFS) Abstraction

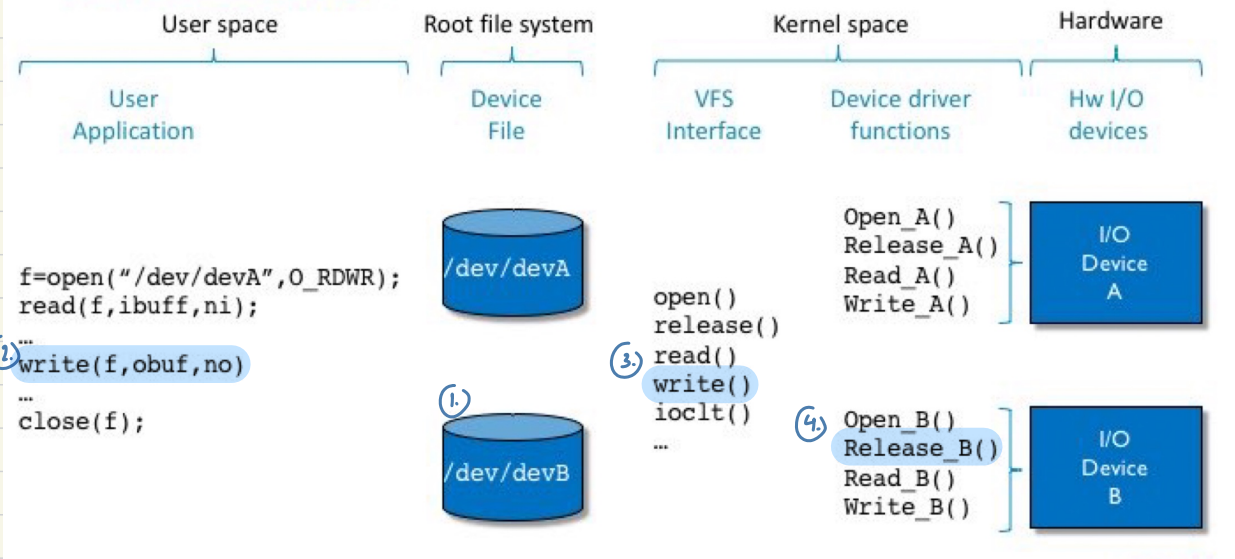
- character / block device จะถูกมองเข้าถึงได้เหมือนกับ file ที่ถูกเก็บไว้ใน file system

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
`f=open("/dev/devA",O_RDWR)`	→ `open()`	→ `Open_A()`
(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

- เป็นตัวกลางที่ user application ส่งผ่านแลกเปลี่ยน data กับ device driver
- device file จะไม่ contain any data ในขณะที่มัน contain ตัวอธิบายที่ประกอบด้วยข้อมูลที่เกี่ยวข้องเพื่อใช้ในการระบุ driver

- 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 driver provide an I/O device ที่มีการใช้งานผ่าน VFS abstraction และ located ใน kernel space
- device driver สามารถ linked กับ Linux Kernel และ executed ที่ system bootstrap
- device driver สามารถเป็น kernel module ได้ ซึ่งจะ loaded ที่ runtime ผ่าน suitable system program หลังจาก Linux Kernel ถูก boot

System programs

- `mknod`, to create a device file
- `insmod`, to insert the module in the kernel
- `rmmod`, 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;

struct cdev dummy_cdev;

static char buffer[64];
```

Data structure used to describe the properties of a character device

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");
    alloc_chrdev_region(&dummy_dev, 0, 1, "dummy_dev");
    printk(KERN_INFO "%s\n", format_dev_t(buffer, dummy_dev));
    cdev_init(&dummy_cdev, &dummy_fops);
    dummy_cdev.owner = THIS_MODULE;
    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.

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

```
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)

```
int cdev_add( struct cdev *p, dev_t dev, unsigned  count);
```

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);
    unregister_chrdev_region(dummy_dev, 1);
}
```

It removes the character device from the 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 bufer 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, offser=%d)\n", (int)count, (int)*f_pos );

    return 1;
}
```

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

It returns the number of byte 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 intermediator through which a kernel application can exchange data with a device driver
- ✔ An intermediator through which a user application can exchange data with a device driver
- ○ An intermediator through which a user application can exchange data with the kernel
- ○ A file that hosts the information regarding a specific device