Real-Time Concepts for Embedded Systems

Author: Qing Li with Caroline Yao


CMPBooks
Chapter 12
I/O Subsystem
Outline

- 12.1 Introduction
- 12.2 Basic I/O Concepts
- 12.3 The I/O Subsystem
12.1 Introduction

- All embedded systems include some form of input and output (I/O) operations
- Examples of embedded systems built explicitly to deal with I/O devices:
  - Cell phone, pager, and a handheld MP3 player
- I/O operations are interpreted differently depending on the viewpoint taken and place different requirements on the level of understanding of the hardware details
Introduction (Cont.)

- From the perspective of a *system software developer*
  - I/O operations imply communicating with the device
  - Programming the device to initiate an I/O request
  - Performing actual data transfer between the device and the system
  - Notifying the requestor when the operation completes
  - Must understand
    - the physical properties (e.g. register definitions, access methods) of the device
    - locating the correct instance of the device
    - how the device is integrated with rest of the system
    - how to handle any errors that can occur during the I/O operations
Introduction (Cont.)

- From the perspective of the **RTOS**
  - Locating the right device for the I/O request
  - Locating the right device driver for the device
  - Issuing the request to the device driver
  - Ensure synchronized access to the device
  - Facilitate an abstraction that hides both the device characteristics and specifics from the application developers
From the perspective of an application developer:

- The goal is to find a simple, uniform, and elegant way to communicate with all types of devices present in the system.
- The application developer is most concerned with presenting the data to the end user in a useful way.
Introduction (Cont.)

- This chapter focuses on
  - basic hardware I/O concepts,
  - the structure of the I/O subsystem, and
  - a specific implementation of an I/O subsystem
12.2 Basic I/O Concepts

- The combination of *I/O devices, device drivers, and the I/O subsystem* comprises the overall I/O system in an embedded environment.

- The purpose of the I/O subsystem
  - To hide the device-specific information from the kernel as well as from the application developer
  - To provide a uniform access method to the peripheral I/O devices of the system
I/O Subsystem and The Layered Model

Diagram:
- Application Software
- I/O Subsystem
- Device Drivers
- Interrupt Handlers
- I/O Device Hardware

Generic

Specific Details
12.2.1 Port-Mapped vs. Memory-Mapped I/O and DMA

- All I/O devices must be initialized through *device control registers* which located on the *CPU board* or in the *devices themselves*.

- During operation, the device registers are accessed again and are programmed to process data transfer requests.

- To access these devices, it is necessary for the developer to determine if the device is *port mapped* or *memory mapped*. 
Port-Mapped I/O

The I/O device address space is *separate* from the system memory address space, special processor instructions, such as the **IN** and **OUT** instructions offered.
Memory-Mapped I/O

The device address is part of the system memory address space. You can access by any memory access instructions.

Available for application use

I/O address space 0xFFFF

Reserved for I/O address space

System address space

LCD

Serial Line

0x0000
DMA I/O

Direct memory access (DMA) chips or controllers allow the device to access the memory directly without involving the processor.
12.2.2 Character-Mode vs. Block-Mode Devices

- Character-mode devices
  - Allow for *unstructured data transfers*
  - Data transfers typically take place in serial fashion (one byte at a time)
  - Simple devices (e.g. serial interface, keypad)
  - The driver *buffers* the data in cases where the transfer rate from system to the device is faster than what the device can handle
Character-Mode vs. Block-Mode Devices

- Block-mode devices
  - Transfer data one block at time (1,024 bytes per data transfer)
  - The underlying hardware imposes the block size
  - Some structure must be imposed on the data or transfer protocol enforced
12.3 The I/O Subsystem

- Each I/O device driver can provide a driver-specific set of *I/O application programming interfaces* to the applications.
  - However, each application must be aware of the nature of the underlying I/O device.

- Thus, embedded systems often include an *I/O subsystem* to reduce this implementation-dependence.
The I/O Subsystem (Cont.)

- I/O subsystem defines *a standard set of functions* for I/O operations
  - To hide device peculiarities from applications

- All I/O device drivers conform to and support this function set
  - To provide uniform I/O to applications across a wide spectrum of I/O devices of varying types
# I/O Functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Create</td>
<td>Creates a virtual instance of an I/O device</td>
</tr>
<tr>
<td>Destroy</td>
<td>Deletes a virtual instance of an I/O device</td>
</tr>
<tr>
<td>Open</td>
<td>Prepares an I/O device for use.</td>
</tr>
<tr>
<td>Close</td>
<td>Communicates to the device that its services are no longer required, which typically initiates device-specific cleanup operations.</td>
</tr>
<tr>
<td>Read</td>
<td>Reads data from an I/O device</td>
</tr>
<tr>
<td>Write</td>
<td>Writes data into an I/O device</td>
</tr>
<tr>
<td>ioctl</td>
<td>Issues control commands to the I/O device (I/O control)</td>
</tr>
</tbody>
</table>
I/O Function Mapping

Application -> I/O System:
- Create()
- Open()
- Read()
- Write()
- Close()
- ioctl()
- Destroy()

Device Driver:
- driver_Create()
- driver_Open()
- driver_Read()
- driver_Write()
- driver_Close()
- driver_ioctl()
- driver_Destroy()
C Structure Defining the Uniform I/O API Set

typedef struct
{
    int (*Create)( );
    int (*Open) ( );
    int (*Read)( );
    int (*Write) ( );
    int (*Close) ( );
    int (*Ioctl) ( );
    int (*Destroy) ( );
} UNIFORM_IO_DRV;
Mapping **Uniform I/O API** to **Specific Driver Functions**

```
UNIFORM_IO_DRV ttyOdrv;
ttyOdrv.Create = tty_Create;
ttyOdrv.Open = tty_Open;
ttyOdrv.Read = tty_Read;
ttyOdrv.Write = tty_Write;
ttyOdrv.Close = tty_Close;
ttyOdrv.Ioctl = tty_Ioctl;
ttyOdrv.Destroy = tty_Destroy;
```
Driver Table

- An I/O subsystem usually maintains a uniform I/O *driver table*
  - Associate *uniform I/O calls* with *driver-specific I/O routines*
  - A new driver can be installed to or removed from this driver table
### Uniform I/O Driver Table

<table>
<thead>
<tr>
<th>Driver</th>
<th>Create</th>
<th>Destroy</th>
<th>Open</th>
<th>Close</th>
<th>Read</th>
<th>Write</th>
<th>loctl</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;fei&quot;</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>&quot;tty&quot;</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

```c
int tty_Create()
{
}
```

```c
int fei_Open()
{
}
```
Associating Devices with Device Drivers

- The create() function is used to create a virtual instance of a device.
- The I/O subsystem tracks these virtual instances using the device table.
- Each entry in the device table holds *generic information*, as well as *instance-specific information*. 
The generic part can include the unique name of the device instance and a reference to the device driver.

- A device instance name is constructed using the generic device name and the instance number.
- For example, the device named tty0 implies that
  - This I/O device is a serial terminal device.
  - The first instance created in the system.
The driver-dependent part is a block of memory
- Hold instance-specific data.
- The content of this information is dependent on the driver implementation.
- The driver is the only entity that accesses and interprets this data.

A reference to the newly created device entry is returned to the caller of the create function.
- Subsequent calls to the open and destroy functions use this reference.
Associating Devices with Drivers

Device Table

- **“tty0”**
  - Reference to driver
  - Instance specific data maintained by the driver

- **“fei0”**
  - Reference to driver
  - Instance specific data maintained by the driver

- Reference to driver
  - Instance specific data maintained by the driver

Driver Table

<table>
<thead>
<tr>
<th></th>
<th>Create</th>
<th>Destroy</th>
<th>Open</th>
<th>Close</th>
<th>Read</th>
<th>Write</th>
<th>Iocntl</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>“fei”</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>“tty”</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Points to Remember

- Interfaces between a device and the main processor occur in two ways: *port mapped* and *memory mapped*
- DMA controllers allows data transfer bypassing the main processor
- I/O subsystems must be flexible enough to handle a wide range of I/O devices.
- Uniform I/O hides device peculiarities from applications.
Points to Remember (Cont.)

- The I/O subsystem maintains a *driver table* that associates *uniform I/O calls* with *driver-specific I/O routines*.

- The I/O subsystem maintains a *device table* and forms an association between this table and the driver table.