Chapter 12: I/O Subsystem

12.1 Introduction

All embedded systems include some form of input and output (I/O) operations. These I/O operations are performed over different types of I/O devices. A vehicle dashboard display, a touch screen on a PDA, the hard disk of a file server, and a network interface card are all examples of I/O devices found in embedded systems. Often, an embedded system is designed specifically to handle the special requirements associated with a device. A cell phone, pager, and a handheld MP3 player are a few examples of embedded systems built explicitly to deal with I/O devices.

I/O operations are interpreted differently depending on the viewpoint taken and place different requirements on the level of understanding of the hardware details.

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, and notifying the requestor when the operation completes. The system software engineer must understand the physical properties, such as the register definitions, and access methods of the device. Locating the correct instance of the device is part of the device communications when multiple instances of the same device are present. The system engineer is also concerned with how the device is integrated with rest of the system. The system engineer is likely a device driver developer because the system engineer must know to handle any errors that can occur during the I/O operations.

From the perspective of the RTOS, I/O operations imply locating the right device for the I/O request, locating the right device driver for the device, and issuing the request to the device driver. Sometimes the RTOS is required to ensure synchronized access to the device. The RTOS must 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.

Each perspective is equally important and is examined in this chapter. 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, associated device drivers, and the I/O subsystem comprises the overall I/O system in an embedded environment. The purpose of the I/O subsystem is to hide the device-specific information from the kernel as well as from the application developer and to provide a uniform access method to the peripheral I/O devices of the system. This section discusses some fundamental concepts from the perspective of the device driver developer.

Figure 12.1 illustrates the I/O subsystem in relation to the rest of the system in a layered software model. As shown, each descending layer adds additional detailed information to the architecture needed to manage a given device.

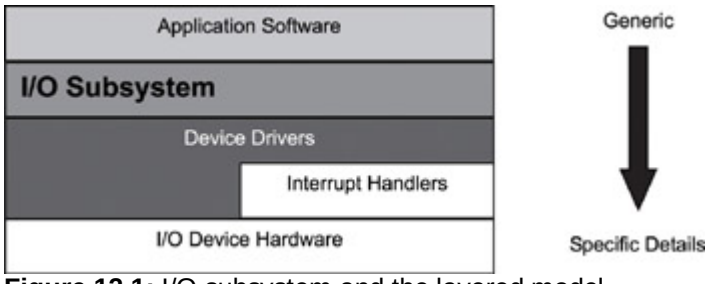


Figure 12.1: I/O subsystem and the layered model.

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

The bottom layer contains the I/O device hardware. The I/O device hardware can range from low-bit rate serial lines to hard drives and gigabit network interface adaptors. All I/O devices must be initialized through device control registers, which are usually external to the CPU. They are 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, which is called *device control*. To access these devices, it is necessary for the developer to determine if the device is port mapped or memory mapped. This information determines which of two methods, port-mapped I/O or memory-mapped I/O, is deployed to access an I/O device.

When 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 by the Intel processor, are used to transfer data between the I/O device and a microprocessor register or memory.

The I/O device address is referred to as the *port number* when specified for these special instructions. This form of I/O is called *port-mapped I/O*, as shown in Figure 12.2.

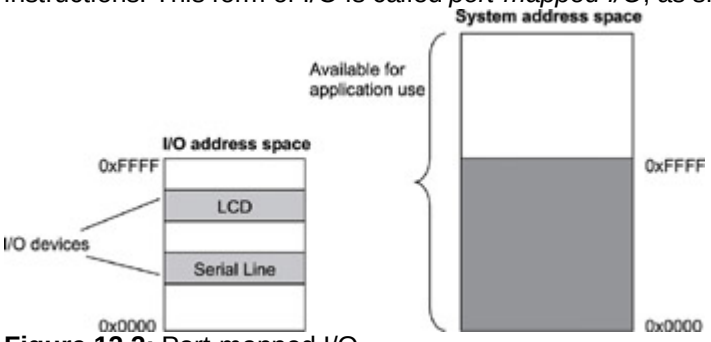


Figure 12.2: Port-mapped I/O.

The devices are programmed to occupy a range in the I/O address space. Each device is on a different I/O port. The I/O ports are accessed through special processor instructions, and actual physical access is accomplished through special hardware circuitry. This I/O method is also called *isolated I/O* because the memory space is isolated from the I/O space, thus the entire memory address space is available for application use.

The other form of device access is memory-mapped I/O, as shown in Figure 12.3. In *memory-mapped I/O*, the device address is part of the system memory address space. Any machine instruction that is encoded to transfer data between a memory location and the processor or between two memory locations can potentially be used to access the I/O device. The I/O device is treated as if it were another memory location. Because the I/O address space occupies a range in the system memory address space, this region of the memory address space is not available for an application to use.

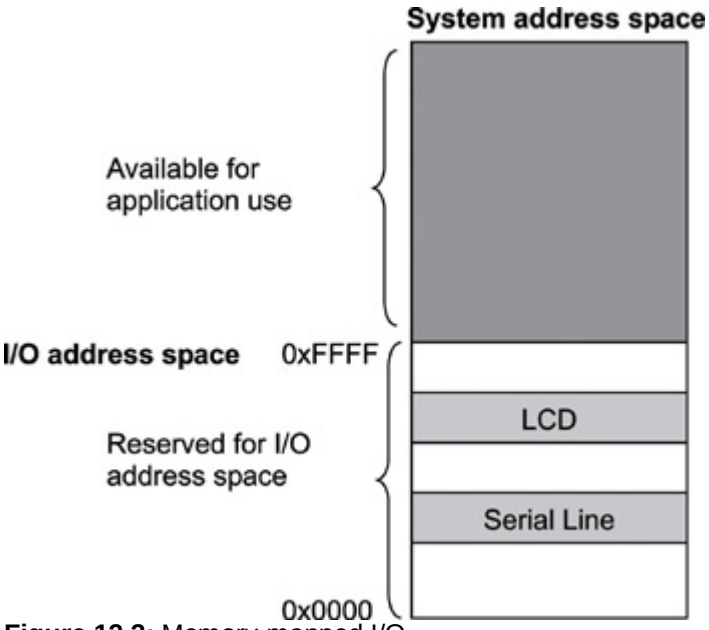


Figure 12.3: Memory-mapped I/O.

The memory-mapped I/O space does not necessarily begin at offset `0` in the system address space, as illustrated in Figure 12.3. It can be mapped anywhere inside the address space. This issue is dependent on the system implementation.

Commonly, tables describing the mapping of a device's internal registers are available in the device hardware data book. The device registers appear at different offsets in this map. Sometimes the information is presented in the 'base + offset' format. This format indicates that the addresses in the map are relative, i.e., the offset must be added to the start address of the I/O space for port-mapped I/O or the offset must be added to the base address of the system memory space for memory-mapped I/O in order to access a particular register on the device.

The processor has to do some work in both of these I/O methods. Data transfer between the device and the system involves transferring data between the device and the processor register and then from the processor register to memory. The transfer speed might not meet the needs of high-speed I/O devices because of the additional data copy involved. Direct memory access (DMA) chips or controllers solve this problem by allowing the device to access the memory directly without involving the processor, as shown in Figure 12.4. The processor is used to set up the DMA controller before a data transfer operation begins, but the processor is bypassed during data transfer, regardless of whether it is a read or write operation. The transfer speed depends on the transfer speed of the I/O device, the speed of the memory device, and the speed of the DMA controller.

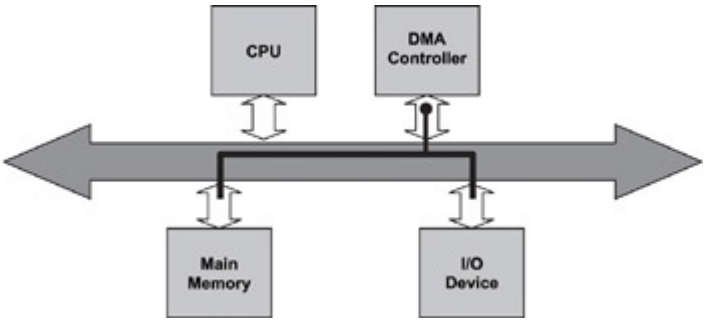


Figure 12.4: DMA I/O.

In essence, the DMA controller provides an alternative data path between the I/O device and the main memory. The processor sets up the transfer operation by specifying the source address, the destination memory address, and the length of the transfer to the DMA controller.

12.2.2 Character-Mode vs. Block-Mode Devices

I/O devices are classified as either character-mode devices or block-mode devices. The classification refers to how the device handles data transfer with the system.

Character-mode devices allow for unstructured data transfers. The data transfers typically take place in serial fashion, one byte at a time. Character-mode devices are usually simple devices, such as the serial interface or the 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.

Block-mode devices transfer data one block at time, for example, 1,024 bytes per data transfer. The underlying hardware imposes the block size. Some structure must be imposed on the data or some transfer protocol enforced. Otherwise an error is likely to occur. Therefore, sometimes it is necessary for the block-mode device driver to perform additional work for each read or write operation, as shown in Figure 12.5.

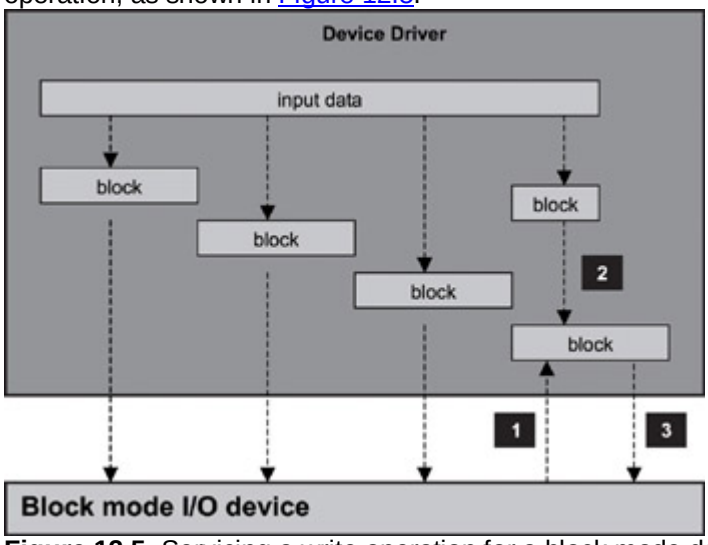


Figure 12.5: Servicing a write operation for a block-mode device.

As illustrated in Figure 12.5, when servicing a write operation with large amounts of data, the device driver must first divide the input data into multiple blocks, each with a device-specific block size. In this example, the input data is divided into four blocks, of which all but the last block is of the required block size. In practice, the last partition often is smaller than the normal device block size.

Each block is transferred to the device in separate write requests. The first three are straightforward write operations. The device driver must handle the last block differently from the first three because the last block has a different size. The method used to process this last block is device specific. In some cases, the driver pads the block to the required size. The example in Figure 12.5 is based on a hard-disk drive. In this case, the device driver first performs a read operation of the affected block and replaces the affected region of the block with the new data. The modified block is then written back.

Another strategy used by block-mode device drivers for small write operations is to accumulate the data in the driver cache and to perform the actual write after enough data has accumulated for a required block size. This technique also minimizes the number of device accesses. Some disadvantages occur with this approach. First, the device driver is more complex. For example,

the block-mode device driver for a hard disk must know if the cached data can satisfy a read operation. The delayed write associated with caching can also cause data loss if a failure occurs and if the driver is shut down and unloaded ungracefully. Data caching in this case implies data copying that can result in lower I/O performance.

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. This arrangement requires each application to be aware of the nature of the underlying I/O device, including the restrictions imposed by the device. The API set is driver and implementation specific, which makes the applications using this API set difficult to port. To reduce this implementation-dependence, embedded systems often include an *I/O subsystem.*

The I/O subsystem defines a standard set of functions for I/O operations in order to hide device peculiarities from applications. All I/O device drivers conform to and support this function set because the goal is to provide uniform I/O to applications across a wide spectrum of I/O devices of varying types.

The following steps must take place to accomplish uniform I/O operations at the application-level.

1. The I/O subsystem defines the API set.
2. The device driver implements each function in the set.
3. The device driver exports the set of functions to the I/O subsystem.
4. The device driver does the work necessary to prepare the device for use. In addition, the driver sets up an association between the I/O subsystem API set and the corresponding device-specific I/O calls.
5. The device driver loads the device and makes this driver and device association known to the I/O subsystem. This action enables the I/O subsystem to present the illusion of an abstract or virtual instance of the device to applications.

This section discusses one approach to uniform I/O. This approach is general, and the goal is to offer insight into the I/O subsystem layer and its interaction with the application layer above and the device driver layer below. Another goal is to give the reader an opportunity to observe how the pieces are put together to provide uniform I/O capability in an embedded environment.

12.3.1 Standard I/O Functions

The I/O subsystem presented in the example in this section defines a set of functions as the standard I/O function set. Table 12.1 lists those functions that are considered part of the set in the general approach to uniform I/O. Again, remember that the example approach is used for illustration purposes in describing and discussing the I/O subsystem in general. The number of functions in the standard I/O API set, function names, and functionality of each is dependent on the embedded system and implementation. The next few sections put these functions into perspective.

Table 12.1: I/O functions.

Function	Description
Create	Creates a virtual instance of an I/O device
Destroy	Deletes a virtual instance of an I/O device
Open	Prepares an I/O device for use.
Close	Communicates to the device that its services are no longer required, which typically initiates device-specific cleanup operations.
Read	Reads data from an I/O device

Table 12.1: I/O functions.

Function	Description
Write	Writes data into an I/O device
Ioctl	Issues control commands to the I/O device (I/O control)

Note that all these functions operate on a so-called 'virtual instance' of the I/O device. In other words, these functions do not act directly on the I/O device, but rather on the driver, which passes the operations to the I/O device. When the open, read, write, and close operations are described, these operations should be understood as acting indirectly on an I/O device through the agency of a virtual instance.

The create function creates a virtual instance of an I/O device in the I/O subsystem, making the device available for subsequent operations, such as open, read, write, and `ioctl`. This function gives the driver an opportunity to prepare the device for use. Preparations might include mapping the device into the system memory space, allocating an available interrupt request line (IRQ) for the device, installing an ISR for the IRQ, and initializing the device into a known state. The driver allocates memory to store instance-specific information for subsequent operations. A reference to the newly created device instance is returned to the caller.

The destroy function deletes a virtual instance of an I/O device from the I/O subsystem. No more operations are allowed on the device after this function completes. This function gives the driver an opportunity to perform cleanup operations, such as un-mapping the device from the system memory space, de-allocating the IRQ, and removing the ISR from the system. The driver frees the memory that was used to store instance-specific information.

The open function prepares an I/O device for subsequent operations, such as read and write. The device might have been in a disabled state when the create function was called. Therefore, one of the operations that the open function might perform is enabling the device. Typically, the open operation can also specify modes of use; for example, a device might be opened for read-only operations or write-only operations or for receiving control commands. The reference to the newly opened I/O device is returned to the caller. In some implementations, the I/O subsystem might supply only one of the two functions, create and open, which implements most of the functionalities of both create and open due to functional overlaps between the two operations.

The close function informs a previously opened I/O device that its services are no longer required. This process typically initiates device-specific cleanup operations. For example, closing a device might cause it to go to a standby state in which it consumes little power. Commonly, the I/O subsystem supplies only one of the two functions, destroy and close, which implements most of the functionalities of both destroy and close, in the case where one function implements both the create and open operations.

The `read` function retrieves data from a previously opened I/O device. The caller specifies the amount of data to retrieve from the device and the location in memory where the data is to be stored. The caller is completely isolated from the device details and is not concerned with the I/O restrictions imposed by the device.

The `write` function transfers data from the application to a previously opened I/O device. The caller specifies the amount of data to transfer and the location in memory holding the data to be transferred. Again, the caller is isolated from the device I/O details.

The `Ioctl` function is used to manipulate the device and driver operating parameters at runtime.

An application is concerned with only two things in the context of uniform I/O: the device on which it wishes to perform I/O operations and the functions presented in this section. The I/O subsystem exports this API set for application use.

12.3.2 Mapping Generic Functions to Driver Functions

The individual device drivers provide the actual implementation of each function in the uniform I/O API set. Figure 12.6 gives an overview of the relationship between the I/O API set and driver internal function set.

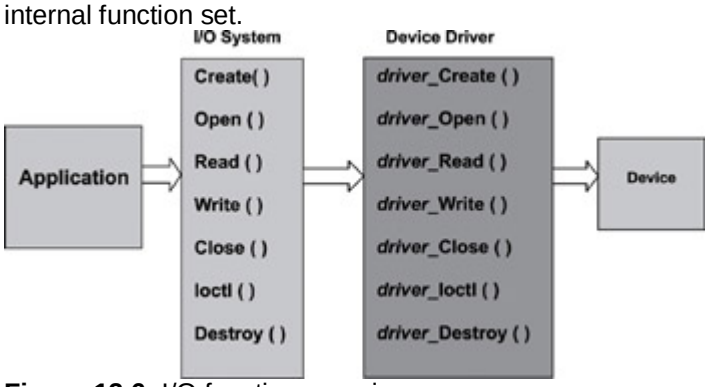


Figure 12.6: I/O function mapping.

As illustrated in Figure 12.6, the I/O subsystem-defined API set needs to be mapped into a function set that is specific to the device driver for any driver that supports uniform I/O. The functions that begin with the *driver_* prefix in Figure 12.6 refer to implementations that are specific to a device driver. The uniform I/O API set can be represented in the C programming language syntax as a structure of function pointers, as shown in the left-hand side of Listing 12.1.

Listing 12.1: 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;
```

The mapping process involves initializing each function pointer with the address of an associated internal driver function, as shown in Listing 12.2. These internal driver functions can have any name as long as they are correctly mapped.

Listing 12.2: Mapping uniform I/O API to specific driver functions.

```
UNIFORM_IO_DRV ttyIOdrv;
ttyIOdrv.Create = tty_Create;
ttyIOdrv.Open = tty_Open;
```

```
ttyIOdrv.Read = tty_Read;
ttyIOdrv.Write = tty_Write;
ttyIOdrv.Close = tty_Close;
ttyIOdrv.Ioctl = tty_Ioctl;
ttyIOdrv.Destroy = tty_Destroy;
```

An I/O subsystem usually maintains a *uniform I/O driver table*. Any driver can be installed into or removed from this driver table by using the utility functions that the I/O subsystem provides. Figure 12.7 illustrates this concept.

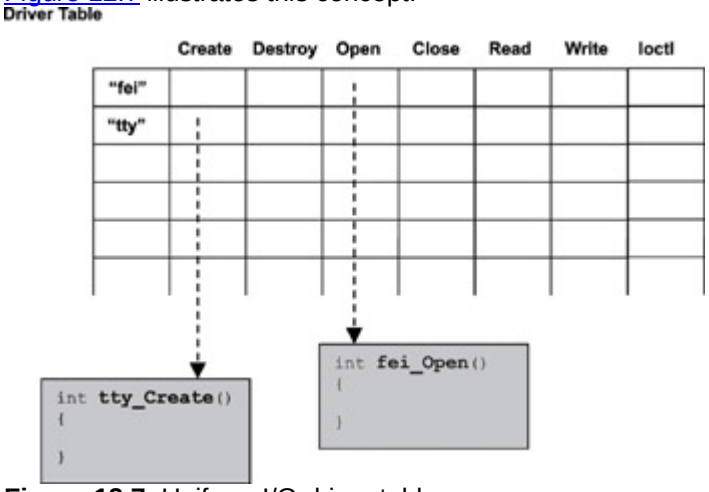


Figure 12.7: Uniform I/O driver table.

Each row in the table represents a unique I/O driver that supports the defined API set. The first column of the table is a generic name used to associate the uniform I/O driver with a particular type of device. In Figure 12.7, a uniform I/O driver is provided for a serial line terminal device, `tty`. The table element at the second row and column contains a pointer to the internal driver function, `tty_Create()`. This pointer, in effect, constitutes an association between the generic create function and the driver-specific create function. The association is used later when creating virtual instances of a device.

These pointers are written to the table when a driver is installed in the I/O subsystem, typically by calling a utility function for driver installation. When this utility function is called, a reference to the newly created driver table entry is returned to the caller.

12.3.3 Associating Devices with Device Drivers

As discussed in the section on standard I/O functions, 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.* A newly created virtual instance is given a unique name and is inserted into the device table, as shown in Figure 12.8. Figure 12.8 also illustrates the device table's relationship to the driver table.

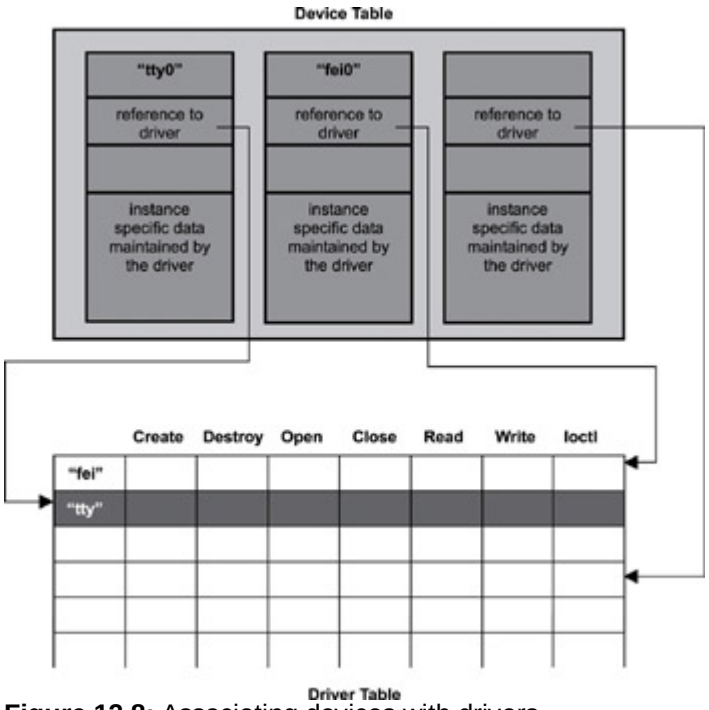


Figure 12.8: Associating devices with drivers.

Each entry in the device table holds generic information, as well as instance-specific information. The generic part of the device entry can include the unique name of the device instance and a reference to the device driver. In Figure 12.8, a device instance name is constructed using the generic device name and the instance number. The device named tty0 implies that this I/O device is a serial terminal device and is the first instance created in the system. The driver-dependent part of the device entry is a block of memory allocated by the driver for each instance to hold instance-specific data. The driver initializes and maintains it. 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.

12.4 Points to Remember

Some points to remember include the following:

- 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.
- 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.