# A DEVICE MANAGEMENT FRAMEWORK FOR AN OBJECT-ORIENTED OPERATING SYSTEM

BY

PANAGIOTIS KOUGIOURIS

Dipl., University of Patras, 1989

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 1991

Urbana, Illinois

# Dedication

To my parents, for their *support* and to $A\rho\gamma\upsilon\rho\grave{\omega}$, for her *patience* and *understanding*.

# Acknowledgements

I want to thank my advisor **Roy H. Campbell** for his comments and support. I would also like to thank **Bjorn A. Helgaas**, **Peter W. Madany** and **Vincent F. Russo** who helped me understand *Choices*. Special thanks to **Ralph E. Johnson** who taught me object-oriented programming and extensively edited a draft of this thesis. Other members of the *Choices* group including **John L. Coolidge**, **David W. Dykstra**, **Nayeem Islam**, **Steven A. March**, **David K. Raila**, **Aamod Sane**, **See-mong Tan** and **Johnathan M. Zweig** helped me in several occasions.

I also want to thank **IBM** and especially my summer 1991 colleagues at Kingston, New York for their support and encouragement. Finally, I want to thank the participants in the talks I gave at **Sun Microsystems** and **Apple Computer** in Spring 1991 for their questions and comments.

# Table of Contents

Chapter


1 Introduction . . . . . . 1

2 Object-Oriented Programming . . . . . . 3
  2.1 Object-Oriented Analysis, Design and Languages . . . . . . 3
  2.2 Object-Oriented Programming Features . . . . . . 4
    2.2.1 Objects . . . . . . 4
    2.2.2 Classes . . . . . . 6
    2.2.3 Inheritance and Polymorphism . . . . . . 7
  2.3 Object-Oriented Design Techniques . . . . . . 8
    2.3.1 Abstract Classes . . . . . . 8
    2.3.2 Object-Oriented Frameworks . . . . . . 9
  2.4 Summary . . . . . . 10

3 The *Choices* Operating System . . . . . . 12
  3.1 Memory Management System . . . . . . 12
  3.2 Process Management . . . . . . 15
  3.3 The Application Programmer Interface . . . . . . 16
  3.4 Summary . . . . . . 17

4 Architectural Support for I/O . . . . . . 18
  4.1 Input/Output Controllers . . . . . . 18
    4.1.1 I/O Devices . . . . . . 19
    4.1.2 Controllers . . . . . . 20
  4.2 Addressing an I/O Controller . . . . . . 21
    4.2.1 Memory-Mapped I/O Address Space . . . . . . 21
    4.2.2 Separate I/O Address Space . . . . . . 22
  4.3 Synchronization Among I/O Controllers and the System . . . . . . 23
  4.4 Programmed and DMA Data Transfer . . . . . . 24
  4.5 Summary . . . . . . 26

5 *Choices* Device Management . . . . . . 27
  5.1 The *DevicesController* Class . . . . . . 28
    5.1.1 Construction of a *DevicesController* . . . . . . 29
    5.1.2 Initialization of a *DevicesController* . . . . . . 29
    5.1.3 Attaching and Detaching *Devices* . . . . . . 31

5.1.4 Sending Commands to a Controller . . . 34
5.1.5 The *Command* Class . . . 35
5.1.6 Interrupt Handling . . . 36
5.2 The *Device* Class . . . 37
5.2.1 The Functionality of a Device . . . 38
5.2.2 The *Choices* Conversion Mechanism . . . 38
5.2.3 Conversion of *Devices* . . . 39
5.3 Configuration of Device Drivers . . . 40
5.4 Summary . . . 41

6 A Reusable Disk Driver . . . 44
6.1 The Machine Dependent Disk Controller Class . . . 44
6.1.1 Sending Commands to a Disk Controller Class . . . 45
6.2 The DiskDevice Class . . . 48
6.2.1 Constructing a DetailedDiskDevice . . . 51
6.2.2 Translating a Disk Block Address . . . 51
6.2.3 Methods of a DetailedDiskDevice . . . 52
6.3 A Detailed Example . . . 52
6.4 The Disk Driver and the *Choices* File System . . . 55
6.5 Summary . . . 55

7 Reusable Character Device Drivers . . . 58
7.1 The Controllers of Character Devices . . . 58
7.2 The Class CharacterDevice . . . 60
7.2.1 Sending Characters to a Device . . . 61
7.2.2 Receiving Characters from a Device . . . 62
7.2.3 Controlling the Output to a CharacterDevice . . . 63
7.3 The Character Device Stream Classes . . . 63
7.3.1 The CharacterDeviceInputStream Class . . . 64
7.3.2 The CharacterDeviceOutputStream Class . . . 64
7.4 Summary . . . 65

8 Summary and Conclusions . . . 67
8.1 Further Issues . . . 69

Bibliography . . . 70

# List of Tables


2.1 The protocol of an object representing the PS/2 processor. . . . . . . . . . . . . 5
2.2 The protocol of the i386CPU class. . . . . . . . . . . . . . . . . . . . . . . . . . 6

3.1 The protocol of the NameServer class. . . . . . . . . . . . . . . . . . . . . . . . 16

4.1 A number of 32-bit system buses and some of their characteristics. . . . . . . . . 24

5.1 The protocol of the *DevicesController* class. . . . . . . . . . . . . . . . . . . . 29
5.2 The protocol of the *Device* class. . . . . . . . . . . . . . . . . . . . . . . . . . 37
5.3 The protocol of the DevicesManager class. . . . . . . . . . . . . . . . . . . . . . 41

6.1 The protocol of the DiskDevice class. . . . . . . . . . . . . . . . . . . . . . . . . 48
6.2 The protocol of the class DetailedDiskDevice. . . . . . . . . . . . . . . . . . . . . 51

7.1 The protocol of the CharacterDevice class. . . . . . . . . . . . . . . . . . . . . . 61
7.2 The protocol of the CharacterDeviceInputStream class. . . . . . . . . . . . . . . . . 63
7.3 The protocol of the CharacterDeviceOutputStream class. . . . . . . . . . . . . . . 64

# List of Figures


2.1 The hierarchy of the CPU classes in *Choices*. . . . . . . . . . . . . . . . . . . . . 7
2.2 The *CPU* abstract class. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 8
2.3 The i386CPU concrete subclass of the *CPU* abstract class. . . . . . . . . . . . . . 9

3.1 The Choices subsystems in the PS/2 implementation. The $y$ axis shows lines of source code. The first bar in each system is code files and the second bar is header files. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 13
3.2 The three layers and five subsystems of the *Choices* Memory Management System 14
3.3 A typical scenario on a *Choices* multiprocessor implementation. The two CPUs share a round-robin scheduler which is a subclass of the *ProcessContainer*. The third one has a special deadline scheduler and is devoted to real-time processes. . 15
3.4 The Domain of a *Choices* application. ObjectProxies reside inside the kernel but are readable by the application. . . . . . . . . . . . . . . . . . . . . . . . . . . 17

4.1 The three components of a computer system. . . . . . . . . . . . . . . . . . . . . . 19
4.2 In this example there are three controllers, namely A,B and C that control two, one and three devices respectively. . . . . . . . . . . . . . . . . . . . . . . . . . . 20
4.3 (a) The Sun's SBus memory-mapped architecture. (b) The IBM's Micro Channel I/O architecture uses a 16-bit I/O address space. . . . . . . . . . . . . . . . . . . 21

5.1 A part of the extensible *DevicesController* hierarchy. . . . . . . . . . . . . . . . . 28
5.2 *DevicesControllers* are at the bottom of the kernel. They interact only with their *Devices*, the DevicesManager and the interrupt handling subsystem. . . . . . . . . 30
5.3 An outline of the `initialize()` method. . . . . . . . . . . . . . . . . . . . . . . . 32
5.4 A skeleton for `DevicesController::attach()` and a concrete subclass . . . . . 33
5.5 The lightweight class *Command*. . . . . . . . . . . . . . . . . . . . . . . . . . . . 35
5.6 The logical structure of the objects participating in the exception management in *Choices*. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 36
5.7 A SerialLine is converted to an *InputStream*. The double dispatching mechanism finds that the SerialLineInputStream class is the *InputStream* that supports SerialLines. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 39
5.8 An outline of the `DevicesManager::addDriver()` method. . . . . . . . . . . . . 42
5.9 An example of *Devices* existing in the *Choices* kernel name server on the PS/2 computer. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 43

6.1 The *DiskCommand* class and its subclasses . . . . . . . . . . . . . . . . . . . . . 46
6.2 An I/O request waiting to be executed by the disk controller. . . . . . . . . . . . 47

6.3 An example of a contiguous section of memory in a virtual memory address space mapped to a discontiguous section in the physical memory address space. . . . . 49
6.4 The method `DiskDevice::fullDiskAddress()`. . . . . . . . . . . . . . . . . . . . 52
6.5 An example that shows how I/O requests are moved inside a system consisting of a DiskController object and two DiskDevices. . . . . . . . . . . . . . . . . . . . 53
6.6 An example that shows how the disk subsystem integrates with other parts of the kernel system. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 55
6.7 An example of a number of disk related commands from the *Choices* `FiSh` on the PS/2 implementation. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 56

7.1 An overview of the Character Input/Output System. . . . . . . . . . . . . . . . . 59
7.2 A number of commands used in the interface between the CharacterDevice and its *DevicesController*. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 60
7.3 An example of inspection of a CharacterDevice object representing a serial line. . 65

# Chapter 1

# Introduction

In this thesis I propose a design for the device management in the *Choices* operating system. *Choices* is an object-oriented operating system written in C++. It was originally implemented on the Encore Multimax, which is a shared memory multiprocessor. The original design and implementation was then ported to workstations and high-end personal computers. The use of object-oriented techniques has made the process, virtual memory and file systems very portable. However, porting the Input/Output system is still a difficult task.

There are two reasons why the porting of the Input/Output subsystem is difficult compared with the other subsystems of *Choices*. The first reason is inherent to the Input/Output system itself. Every system programmer that has ever written or modified a device driver knows how difficult is to communicate with the Input/Output hardware. Every Input/Output architecture has its own pecularities. The second reason is that the original implementation of *Choices* (on the Encore Multimax), used firmware support to communicate with the devices. The firmware support gives the programmer an ideal "picture" of the hardware. However, such support is usually not existent in all the hardware platforms. When it exists it is not usually used by general purpose operating systems because it imposes performance overheads and is very inflexible.

In this thesis I propose an object-oriented framework for the device management in *Choices*. The device management forms the low level of the *Choices* Input/Output subsystem. The motivation for this framework is that there are many software components inside a device driver that can be reused. Such components can be usually reused among drivers for the same type of devices in the same or different computers. For instance, disk scheduling algorithms are implemented as part of the disk driver because they have to know about the details of the disk. However, such algorithms can be reused for different disk drivers. Furthermore, such algorithms should not be part of the device driver to seperate policy and mechanism[43].

The proposed framework separates the functions that a device offers from the higher level abstractions that other parts of the system expect. As a result, an implementor of a device driver can concentrate on the details of the hardware. The job of splitting high-level Input/Output "commands" to primitive commands, which the hardware can execute, is assigned to other reusable objects. The design I describe here is the result of a number of prototypes that were implemented on the IBM PS/2 implementation of *Choices*. One of these prototypes was also used in the *Choices* implementation on the Apple Macintosh IIx computer.

This thesis has two parts. The first part consists of the chapters 2–4 and contains introductory material. The second part consists of the chapters 5–8 and contains the description of the framework. The second chapter introduces some object-oriented programming terms. Even if you are a wizard in object-oriented programming, I would advise you not skip this chapter because establishes a terminology and notation that I use in the rest of the thesis. The third chapter is a brief introduction to the *Choices* operating system. The fourth chapter discusses some Input/Output issues that are common across different architectures.

The fifth chapter, which is the first chapter of the second part, describes the overall framework. The framework is based on two hierarchies of classes. The one is the class *DevicesController* and the other is the class *Device*. The proposed way for configuring the devices of a system is also described in the fifth chapter.

The sixth chapter gives an example of reusable classes that can be used for the implementation of a disk device driver. The seventh chapter gives an example of reusable classes that can be used for the implementation of a device driver for a character device—like a keyboard or a serial line. Finally, the last chapter summarizes this thesis and discusses some further issues.

# Chapter 2

# Object-Oriented Programming

One of the recent advances in the area of software engineering is the use of "object-oriented" techniques in the development of software systems. Although this particular style of software development has been around for more than 20 years[12], there is still confusion on what "object-oriented" means and a lack of common terminology and notation. In this chapter I introduce object-oriented terms in order to establish a terminology and notation for the rest of the thesis. The terminology I use is influenced by the SMALLTALK programming language[20] and the notation by the C++ language[17].

## 2.1 Object-Oriented Analysis, Design and Languages

Object-oriented programming is a particular kind of software development. It is based on the observation that the problem domain of most programs is composed of interacting objects. The problem domain of a program is the environment in which the program solves a problem. For instance, the program domain of an air-traffic control program consists of airplanes, airports, runways, gates and other **key abstractions**. The process of figuring out the key abstractions of a domain and the way they are related is an art rather than a science and implies an understanding of the application domain. This process is called **object-oriented analysis**[10].

Object-oriented analysis helps one determine the objects of a problem domain. However, not all the abstractions of the problem domain are important. Objects with the same behavior belong to the same **class**. Since some classes inherit the behavior of other classes **hierarchies** of classes are formed. The process of classifying objects into classes, putting classes into hierarchies and explaining how the objects interact to solve a problem is called **object-oriented design**[5, 52].

Object-oriented analysis and design lead to software architectures that can be easily implemented by using object-oriented programming languages. Object-oriented programming languages are languages that support objects, classes, inheritance[51], polymorphism and late binding. In this sense C++, Smalltalk and Eiffel are all object-oriented languages. On the other hand, languages like Ada that support abstract data-types but do not support inheritance and late binding are not considered object-oriented languages.

## 2.2 Object-Oriented Programming Features

In this section I introduce several features of object-oriented programming. These features are found in some form in all the object-oriented programming languages. As I mentioned in the introduction of this chapter I discuss these features to establish a common terminology and notation which I use throughout this thesis.

### 2.2.1 Objects

An object is the most fundamental concept of an object-oriented language. An object can be accessed only by sending **messages**[1] to it. Each message consists of a **selector** and a number of arguments. Sending a message to an object leads to the execution of a **method**. Each method is related to a **signature**. A signature is the name of the selector a method and the types of its input and output arguments[52]. Different methods of different objects can share the same signature. The set of signatures that can be used as patterns for messages sent to an object is called the **protocol** of the object. Attempts to send an arbitrary message to an object are handled by either the compile-time or the run-time system of the language.

Objects have **state**. Every time a method is invoked on an object the behavior of the object depends on the state of the object. The invocation of a method can move an object from one state to another. Objects are usually created in a known state by the **constructor** of the object.

[1] Messages in an object-oriented system should not be confused with messages in a distributed system. A message in an object-oriented system usually results in a function call.

| Intel386 CPU object | | |
|---|---|---|
| `ExceptionManager *` | `installExceptionManager` | `( ExceptionManager * )` |
| `Status` | `enableInterrupts` | `()` |
| `Status` | `disableInterrupts` | `()` |
| `Process *` | `currentProcess` | `()` |
| `void` | `setIdleContainer` | `()` |
| `void` | `preempt` | `()` |
| `TimeSliceTimer *` | `timeSliceTimer` | `()` |
| `AddressTranslator *` | `MMU` | `()` |
| `unsigned long` | `id` | `()` |

Table 2.1: The protocol of an object representing the PS/2 processor.

In languages like Smalltalk that support garbage collection, objects are implicitly deleted when they are not used any more. In other languages, like C++, objects must be explicitly deleted.

**Example 2.1** In the PS/2 implementation of *Choices*, the system's CPU is represented as an object. The protocol of this object is shown in Table 2.1. The **`installExceptionManager()`** method installs a new *ExceptionManager*, which is responsible for handling exceptions generated by programs running on this CPU and interrupts generated by devices attached to this CPU. The methods **`enableInterrupts(), disableInterrupts()`** enable and disable interrupts to the CPU. The method **`currentProcess()`** returns the process that is running on this CPU. The method **`setIdleContainer()`** replaces the CPU's idleContainer. The idleContainer is a reference to a list of processes *ready* to run. By **`preempting`** the CPU, the current process is added back to the *idleContainer* and the next ready process is dispatched. The rest of the methods provide access to encapsulated objects related to this CPU, like its timeSliceTimer, its Memory Management Unit and its identifier.

Table 2.1 presents the signatures in a C++ header file format. Every signature consists of three components. The first is the type of the value it returns. The second is the name of the selector. Finally, the third element is a tuple consisting of the parameters that should be provided when a message following the pattern of this signature is sent. For example, the first signature in Table 2.1 has a selector called **`installExceptionManager`** which takes an *ExceptionManager* object as an argument and returns another *ExceptionManager*, which happens to be the old manager. In this thesis, each selector uniquely identifies a signature

| CLASS: i386CPU | SUPER: - | |
|---|---|---|
| `ExceptionManager *` | `installExceptionManager` | `( ExceptionManager * )` |
| `Status` | `enableInterrupts` | `()` |
| `Status` | `disableInterrupts` | `()` |
| `Process *` | `currentProcess` | `()` |
| `void` | `setIdleContainer` | `()` |
| `void` | `preempt` | `()` |
| `TimeSliceTimer *` | `timeSliceTimer` | `()` |
| `AddressTranslator *` | `MMU` | `()` |
| `unsigned long` | `id` | `()` |

Table 2.2: The protocol of the i386CPU class.

so I use the name of the selector to refer to the whole signature. Selectors always appear in `typewriter` font. □

### 2.2.2 Classes

Different objects often have the same functionality. This leads to the introduction of classes. The objects in the same class share the same structure, protocol and implementation. Every object in almost any object-oriented language is an **instance** of a class.

A class is a design abstraction and does not necessarily exist at run-time. However, in *Choices* we realized that having objects that represent classes is very useful and we introduced a mechanism that allows classes as first class objects[30, 36].

**Example 2.2** Table 2.2 shows the protocol of the class i386CPU. The object in the example 2.1 could be an instance of the class i386CPU. A uniprocessor system like the IBM PS/2[26] has only one instance of this class. However a multiprocessor system based on the Intel386 processor would have several i386CPU objects, one for each processor. From now on, when I write *a SomeClass* I mean an instance of the class *SomeClass*.

Table 2.2 shows the way I use to present the protocol of a class. The top box contains the name of the class and the name of its superclass. The bottom box contains the protocol of the class. Signatures inside the bottom box are separated by lines according to the logical category to which they belong. For instance, in Table 2.2, the first category contains signatures related to interrupt and exception handling. The second category contains signatures related to process scheduling and dispatching. The third category contains accessors. □

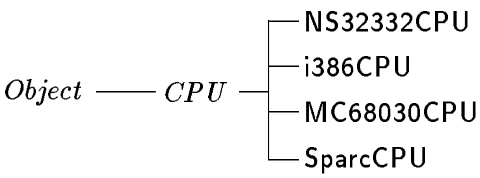


Figure 2.1: The hierarchy of the CPU classes in *Choices*.

### 2.2.3 Inheritance and Polymorphism

One of the most important features of an object-oriented language is that a class can **inherit** structure and implementation from another class. The first class is called a **subclass** of the second class, which is called its **superclass**. The subclass inherits the protocol of the superclass. However, the implementation of some methods might be different and the subclass might extend the protocol. When a subclass changes the implementation of a method defined in the superclass, we say that the subclass **overloads** the method of the superclass. Otherwise the subclass **inherits** the method of the superclass.

Inheritance leads to hierarchies of classes[2]. In a hierarchy, the class that does not have any superclasses is called the **root class** of the hierarchy. The classes in the path from the root class to a class *SomeClass* in a hierarchy are called the **ancestors** of the class *SomeClass*. All the classes in the hierarchy rooted by a class *SomeClass*—except the class *SomeClass* itself—are called the **descendants** of the class. For instance, all classes, except the root class, are **descendants** of the root class of a hierarchy.

Since a subclass shares the protocol of its superclass, an instance of a descendant class of a class *SomeClass* can be used wherever an instance of the class *SomeClass* can be used. Object-oriented languages allow names that can store entities of more that one type. For example, in C++, an identifier that has been defined as a pointer to an object of class *SomeClass* can point to any object which is an instance of *SomeClass* or an instance of a descendant class of *SomeClass*. This feature of a language is called **polymorphism**[7].

**Example 2.3** Figure 2.1 shows the hierarchy of the processor classes in *Choices*. In this hierarchy *Object* is the root class. Since all CPUs inherit from the class *Object*, they inherit the

[2]Some languages allow a class to have multiple superclasses. Such a feature allows directed acyclic graphs of classes. In *Choices* we decided that using multiple inheritance leads to space and time inefficiencies, so we don't use this feature.

| CLASS: *CPU* | SUPER: *Object* | |
|---|---|---|
| `ExceptionManager *` | `installExceptionManager` | `( ExceptionManager * )` |
| `Status` | `enableInterrupts` | `() = 0` |
| `Status` | `disableInterrupts` | `() = 0` |
| `Process *` | `currentProcess` | `()` |
| `void` | `setIdleContainer` | `()` |
| `void` | `preempt` | `() = 0` |
| `TimeSliceTimer *` | `timeSliceTimer` | `()` |
| `AddressTranslator *` | `MMU` | `()` |
| `unsigned long` | `id` | `() = 0` |

Figure 2.2: The *CPU* abstract class.

structure and the protocol of this class. For example, the method **`printName()`** defined in class *Object* can be invoked on any *CPU*. The i386CPU class is a subclass of the *CPU* class. *CPU* is the superclass of four classes. The machine independent code of *Choices* uses *CPU* objects. Depending on the target platform, the object can belong to any concrete subclass of *CPU*.□

Overloading leads to a number of different methods using the same signature. A message, in the source of an object-oriented program, does not reveal which implementation will be executed when this message is sent at run-time. The implementation depends on the object that receives the message. Since the class of an object referenced by a variable might not be known until run-time, the decision of which implementation to execute is deferred until then. This run-time decision is called **late binding**.

## 2.3 Object-Oriented Design Techniques

This section presents two important object-oriented design techniques. The first is the use of abstract classes, that is classes that exist only for inheritance. The second is the use of object-oriented frameworks, that is reusable object-oriented designs.

### 2.3.1 Abstract Classes

Classes that do not have any instances but exist only for inheritance purposes are called **abstract classes**[32, 31]. An abstract class makes a number of assumptions about objects in the application domain. These assumptions lead to a structure, a protocol and (typically) a partial implementation. Subclasses of the abstract class must follow these assumptions because they

| CLASS: i386CPU | SUPER: *CPU* | |
|---|---|---|
| `ExceptionManager *` | `installExceptionManager` | `( ExceptionManager * )` |
| `Status` | `enableInterrupts` | `()` |
| `Status` | `disableInterrupts` | `()` |
| `void` | `preempt` | `()` |
| `unsigned long` | `id` | `()` |

Figure 2.3: The i386CPU concrete subclass of the *CPU* abstract class.

inherit the protocol and the structure. These assumptions should be stated clearly in the documentation of the class, to make the abstract class useful. In particular, the way the abstract class is supposed to be used by subclasses should be made explicit.

Abstract classes are very important because they help the decomposition of a system. In addition, they are a way of reusing design and code. A well designed abstract class will become part of an object-oriented framework—see 2.3.2—and will be reused by different applications.

Some methods of an abstract class have no implementation. These methods are called **abstract**. Concrete subclasses should provide implementation for these selectors. For example, in Figure 2.2 the method `preempt()` of the class *CPU* is an abstract method. However, the method `setExceptionManager()` of the same class is not abstract. Such fully implemented methods of an abstract class are called **base** methods. Finally, methods of an abstract class that are implemented in terms of abstract methods are called **template** methods[32].

In my C++ like notation, the name of an abstract class appears in *italics*. Classes that are not abstract are called **concrete**. In this thesis concrete classes appear appear in Sans Serif. Finally, the selector of an abstract method is followed by an `= 0`[3].

### 2.3.2 Object-Oriented Frameworks

The classes and the hierarchies of an object-oriented system show only part of the design. For instance, the way these classes are supposed to interact is a very important aspect of an object-oriented system not addressed by a description based only on the classes of the system. The classes describe the types of building blocks of the system but do not describe how these blocks are supposed to be plugged together. Booch[5] lists two kind of relations—using and containing—among objects, and four kind of relationships—inheritance, using, instantiation

[3] This is the way to denote a *pure virtual function* in C++ .

and metaclass—among classes in an object-oriented system. Very few of these relations are clear by examining only the hierarchies and the protocols of the classes of the design.

Another issue that should be addressed by an object-oriented design is the model of the application domain that led to the classes in the hierarchies. A number of class hierarchies, along with the assumptions of the model and the way these classes should be instantiated and plugged together, form an **object-oriented framework**. There are many formal ways to describe the classes of an object-oriented system. However, up to now, there is no widely used formal way to describe how objects are supposed to interact. Research is being conducted in this direction[24, 52].

Object-oriented frameworks are very important in the design of a system. However, their real power is that they can be reused. Since a framework is an abstraction of a problem domain, it can be reused by other applications in the same domain. *Choices* frameworks are excellent examples of reusability. The problem domain of *Choices* is the area of operating systems. The frameworks designed for the Encore Multimax architecture were later reused to build operating systems for other architectures.

There is a trade-off between the number of assumptions a framework makes and the flexibility and reusability of the framework. For example, the *Choices* virtual memory and memory management frameworks make the assumption that the underlying architecture has at least two hardware enforced protection levels. This assumption makes porting *Choices* to some architectures impossible, unless a major framework reorganization takes place. A lot of experience in the application domain and vision is needed to come up with a reusable framework the very first time. In practice, the life-cycle of most frameworks consists of a number of reorganizations[32, 39].

## 2.4 Summary

This chapter introduced some object-oriented terms to establish a common terminology and notation. Objects are instances of classes. Each class has a protocol consisting of a number of signatures. Concrete classes have a method associated with each signature in their protocol. Abstract classes do not have a method associated with each signature. Therefore, they cannot be instantiated. Subclasses inherit the structure, the protocol and part of the implementation of their superclasses. Hierarchies of classes along with an informal description on how they can be used form an object-oriented framework.

This thesis presents a framework for the device management subsystem of the *Choices* operating system. A prototype based on this framework has been used in the implementation of *Choices* on the PS/2 computer. I tried to make this framework reusable by keeping in mind not only the PS/2 architecture but also the architecture of other high-end PCs and workstations like the Macintosh IIx and the Sun SPARCStation. The prototype has been reused in the Macintosh implementation of *Choices* and the results are very encouraging. The next chapter is a brief description of the *Choices* operating system. It describes other frameworks that interact with the device management framework. Chapter 4 describes issues surrounding typical I/O architectures, the problem domain of a device management framework.

# Chapter 3

# The *Choices* Operating System

*Choices*[43] is an object-oriented framework for operating systems on multiprocessor and uniprocessor architectures[32]. The original system was used to implement a native operating system for the Encore Multimax, which is a shared memory multiprocessor architecture. The framework was then reused for implementations on the Apple Macintosh[23], the AT&T6386 and the IBM PS/2 computers. Currently, *Choices* implementations are being made on the Sun SPARCStation 2 and the Intel hypercube. The framework is implemented in C++[17], which was selected as the best available object-oriented language for building an operating system. Each implementation consists of a machine independent, a machine dependent and a processor dependent part. Figure 3.1 shows the size of the *Choices* subsystems for the PS/2 implementation.

In *Choices*, all operating system entities are represented as C++ objects. For instance, all CPUs, MMUs, processes, schedulers, virtual address spaces, physical memory frames, and files are objects in *Choices*. *Choices* objects are instances of about 300 classes. In this chapter I briefly introduce two basic *Choices* subsystems, namely the memory management system and the process management system. I also introduce the *Choices* object-oriented Application Programmer Interface.

## 3.1 Memory Management System

The *Choices* memory management system is divided into five subsystems arranged in three layers[41] as shown in figure 3.2. The Virtual Memory subsystem forms the topmost layer. It

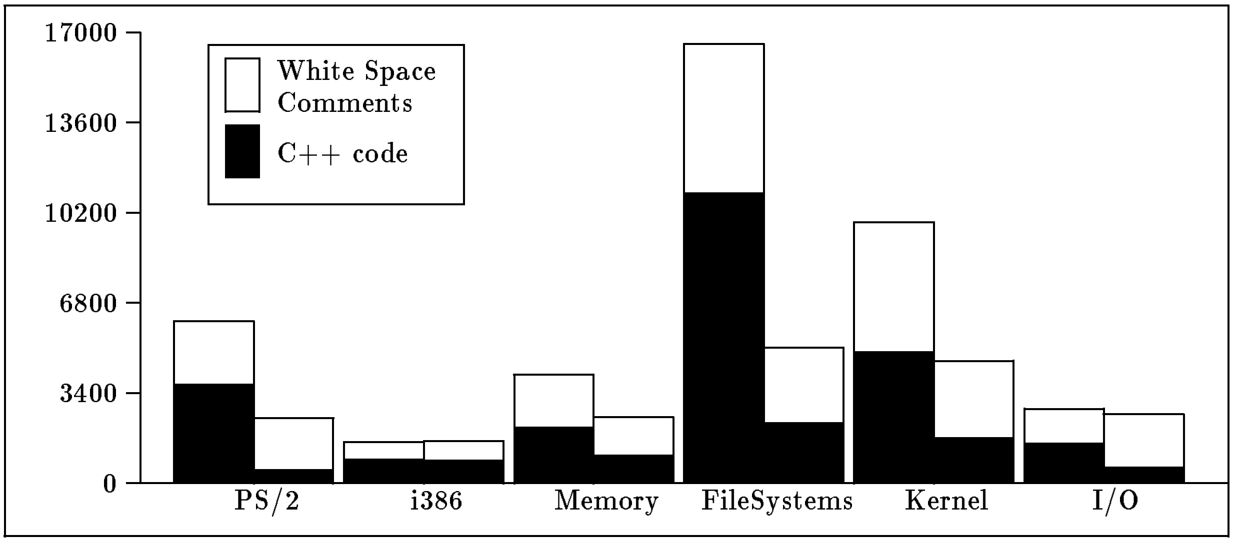


Figure 3.1: The Choices subsystems in the PS/2 implementation. The $y$ axis shows lines of source code. The first bar in each system is code files and the second bar is header files.

is supported by the Address Translation and the Logical Caching subsystems, which coexist in the second layer. The logical caching subsystem, which is built on top of the Physical and Logical Memory Subsystems, forms the third layer.

The **Physical Memory** subsystem is responsible for allocation and deallocation of the physical memory. An instance of the class Store represents the physical memory of the machine. Instances of the class PhysicallyAddressableUnit represent individual physical memory frames. A list of discontiguous physical memory frames is represented as a PhysicalMemoryChain. A PhysicalMemoryChain is usually used as a description for a scatter-gather DMA device(see section 4.4). A PhysicalMemoryChain usually corresponds to a contiguous block of memory in a virtual address space[43].

The **Address Translation** subsystem is responsible for translating virtual addresses to physical addresses by exploiting the memory management hardware. The class *AddressTranslator* is an abstract class that represents Memory Management Units. The class is subclassed by the machine dependent code in each *Choices* implementation. The class *AddressTranslation* provides the rest of the memory management system with a machine independent interface. It corresponds to the page or segment tables that are kept by the architecture. Concrete machine dependent subclasses implement the *AddressTranslation* protocol. *Choices* follows the Mach[18] approach and keeps the address translation tables in a machine independent format.

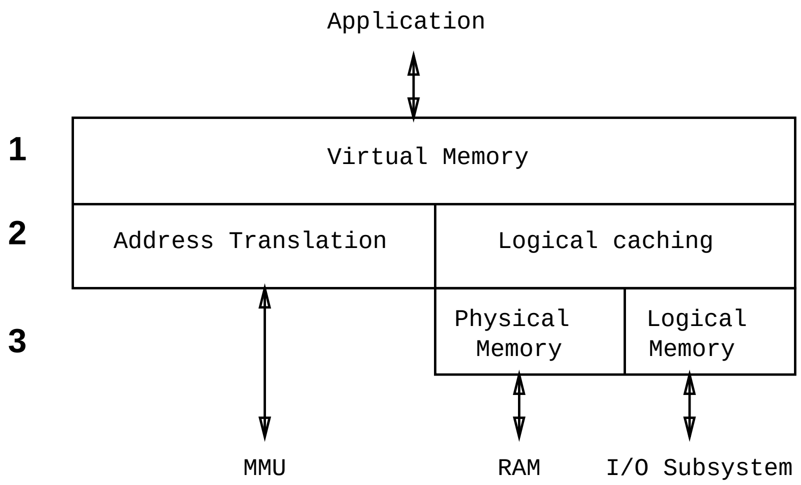


Figure 3.2: The three layers and five subsystems of the *Choices* Memory Management System

The **Logical Memory** subsystem is responsible for managing the backing store. It connects the Memory Management system with the File system and the Input/Output system. The class *MemoryObject* represents abstractions that have a number of randomly accessed, identically sized blocks and transfer blocks from and to the physical memory. Disks, disk-partitions, and files are some common *MemoryObjects*.

The **Logical Caching** subsystem is responsible for caching *MemoryObjects* in the physical memory of the computer. This is necessary for both practical and efficiency reasons. For instance, a *MemoryObject* that corresponds to an executable file cannot be executed unless it is cached in the main memory of the computer. The class *MemoryObjectCache* provides the mechanism for caching. Memory-mapped files are implemented in *Choices* by using this layer[37].

Finally, the **Virtual Memory** subsystem resides on top of the other layers and provides applications with the abstraction of a huge address space. Virtual address spaces in *Choices* are represented by instances of the class Domain. Each Domain has a number of *MemoryObjects* mapped through *MemoryObjectCaches*. Each Domain also has an *AddressTranslation* that is activated on a CPU's *AddressTranslator* as a result of a heavyweight process context switch. The kernel space is mapped into every Domain at the same address. In many architectures this approach makes the invocation of system calls much cheaper. Memory can be shared between a number of Domains by mapping a *MemoryObject* and its *MemoryObjectCache* in

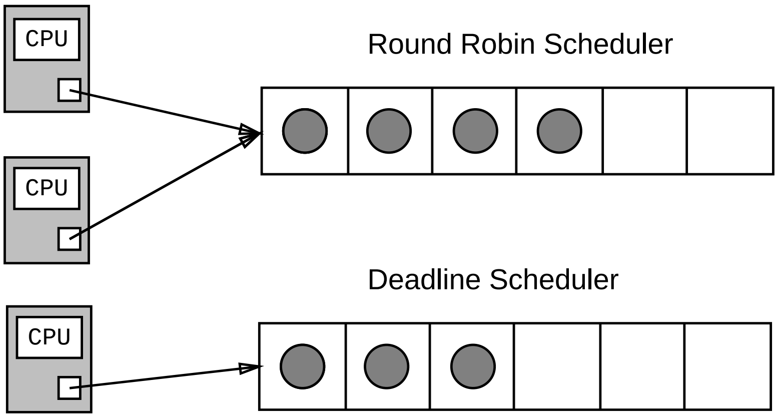


Figure 3.3: A typical scenario on a *Choices* multiprocessor implementation. The two CPUs share a round-robin scheduler which is a subclass of the *ProcessContainer*. The third one has a special deadline scheduler and is devoted to real-time processes.

all the domains. Domains "fix" page faults by sending the **makeAddressable** message to the responsible *MemoryObjectCache* and by changing the *AddressTranslations*.

## 3.2 Process Management

In *Choices*, running programs are represented by instances of the class *Process*. Each *Process* runs inside a Domain. The concepts of process and virtual address space are completely orthogonal[42]. This way many *Processes* can run inside the same Domain leading to so called threads or lightweight processes.

*Processes* run on *CPU* objects. A multiprocessor system has multiple *CPUs* while a uniprocessor has only one. The machine dependent part of a *Process* is kept in a seperate *Context* object. When a process is dispatched the context is "loaded" on the processor. If the process leaving the *CPU* and the process dispatched on the CPU are running in the same Domain, the address translation mappings are not changed, leading to a lightweight context switch.

Processes that are ready to run when there are no available *CPUs* wait in *ProcessContainers* as depicted in figure 3.3. The abstract class *ProcessContainer* provides the interface for the *Choices* schedulers. Concrete subclasses for this class implement different scheduling policies[35].

| NameServer | *ProxiableObject* | |
|---|---|---|
| | `NameServer` | `( rootNameServer * )` |
| | `~NameServer` | `()` |
| `Object *` | `lookup` | `( char * name, Class * )` |
| `void` | `bind` | `( char * name, Object *)` |
| `void` | `unbind` | `( Object * )` |
| `void` | `unbind` | `( char * name )` |

Table 3.1: The protocol of the NameServer class.

Blocked processes usually wait on semaphores. Semaphores is the *Choices* "heavyweight" synchronization primitive. Semaphores are built on top of busy-waiting objects called *Locks*. The *Locks* are machine dependent objects. On uniprocessors the implementation of *Locks* simply disables and enables processor interrupts. On shared memory multiprocessors *Locks* are implemented using hardware atomic fetch-modify instructions[40]. However, the semaphore code is completely machine independent.

## 3.3 The Application Programmer Interface

*Choices* provides an object-oriented Application Programmer Interface (API)[43] which allows application processes to send messages to objects across different protection boundaries[14, 15]. When a *Choices* application is started, it gets a reference to a NameServer(see Table 3.1). A NameServer is associated with each Domain. NameServers are objects that map symbolic names to references to kernel objects. The application then can look up strings in the NameServer and get back kernel objects. For instance, the NameServer contains the entries `"StandardInput"` and `"StandardOutput"` that give access to objects representing the input and output streams of the process.

Although the application programmer gets the impression that he gets the real objects, the objects returned by the kernel are instances of the class to ObjectProxy[45] as depicted in Figure 3.4. Each ObjectProxy consists of a reference to the real object and resides in a portion of the kernel address space that is readable but not writable by the applications.

Sending a message to an ObjectProxy causes a trap to the kernel protection level. The kernel checks that the ObjectProxy reference corresponds to a system object. It then invokes the method on behalf of the application on the system object. A special mechanism allows the

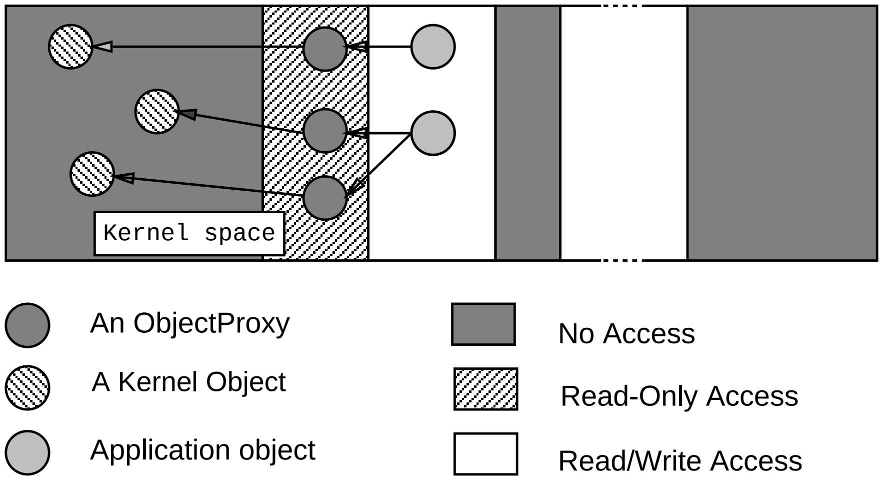


Figure 3.4: The Domain of a *Choices* application. ObjectProxies reside inside the kernel but are readable by the application.

applications to construct new objects that reside in the kernel address space. For instance, this is the way that an application process can build a new process.

## 3.4 Summary

*Choices* is a general purpose object-oriented operating system. It consists of a general operating system framework and a number of subframeworks. The classes *Process*, *MemoryObject* and Domain described in this chapter are used to "glue" all the subframeworks together. An object-oriented Application Programmer's Interface based on *ObjectProxy* and *NameServer* objects provides access to kernel objects and services. Chapters 5,6 and 7 introduce a device management subframework.

# Chapter 4

# Architectural Support for I/O

A modern computer consists of three sets of components, namely: Central Processing Units, memory modules and Input/Output devices(see Figure 4.1). Over the years the speed, instruction set, size and other factors of a CPU have changed but its role has always been to fetch, decode and execute instructions which reside in the main memory of the computer. The main memory consists of a number of memory modules that vary in speed, power consumption, cost and size. Its role is to store and retrieve words under the control of processors of the system. Finally, there are great variety of Input/Output devices. Their role is to move data from and to the main memory of a computer system. In general, very few devices are part of the architecture of a general purpose computer system. The Input/Output architecture defines an "open" interface. Numerous devices adhering to the interface can be connected to the system.

This chapter describes the most important issues of Input/Output. The first section discusses the introduction of Input/Output Controllers between the devices and the system. The second section discusses the different ways to access a device. The third section discusses how the system and the Input/Output controllers get synchronized. Finally, the fourth section describes the two most common ways of transferring data between a controller and the memory of the system.

## 4.1 Input/Output Controllers

The amount of time it takes a computer system to execute a given task has decreased dramatically over the last forty years. However, problems that need more computational power will

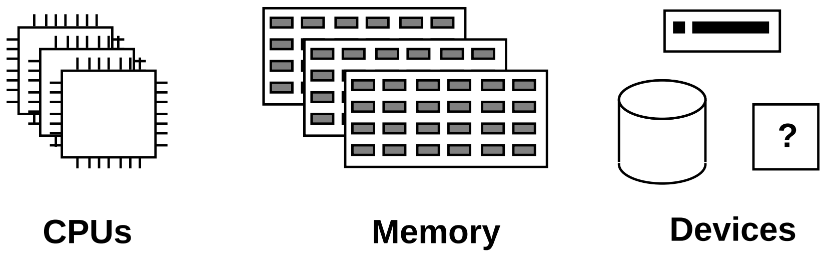


Figure 4.1: The three components of a computer system.

always exist. This is the motivation behind even faster computer architectures. Getting components from a "fastest" hardware technology is one way to provide faster computer systems. However, such a solution is not cost-effective because faster technologies, when they are available, are usually more expensive[46]. The real challenge for a computer architect is to build the fastest computer architecture using a given component technology. The challenge of optimizing an aspect of a system under certain constraints is inherent in the engineering discipline.

### 4.1.1 I/O Devices

For a user a device is a hardware component of a computer system that can perform some operations. For instance a disk, is a random-access device that can store large amounts of information even when the power is off. As another example, a plotter is a device that can move a pen on top of a sheet of paper under software control to draw text or figures. Some devices exist only to provide an interface to other "standard" devices. For instance, serial lines and SCSI ports exist only to connect serial and SCSI devices respectively. An I/O architecture should give the user the ability to connect a large and diverse number of devices to the system.

Internally each device usually consists of two subsystems. The first one is the mechanical parts. In a disk the mechanical parts are the disk plates, the heads and the electromagnetic mechanism that controls them. In a laser printer the mechanical part is the print engine. The second subsystem of a device is hardware and software that controls the mechanical parts and makes the device accessible to the system. However, there are devices that do not have moving parts. For example, a bubble store or an ethernet port are devices that do not have any mechanical parts.

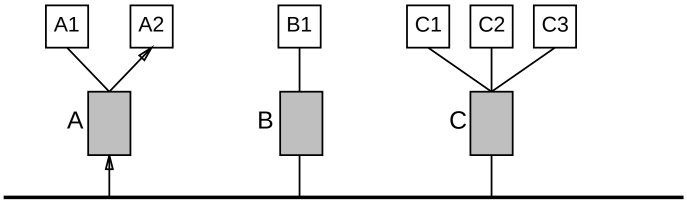


Figure 4.2: In this example there are three controllers, namely A,B and C that control two, one and three devices respectively.

### 4.1.2 Controllers

The device is accessible either directly or indirectly by a CPU. In the latter case a hierarchy of **controllers** exists between the device and the CPU. Each controller is usually shared by a number of devices. An example of a direct connection between a device and a CPU is the console subsystem found in a number of minis and mainframes. This subsystem directly connects a communication line with the CPU, to be used when the operating system is booted. At the other extreme, an example of an indirect connection is the IBM channel organization[21]. In this architecture a disk is connected to a storage director which in turn is connected to a channel. In the IBM channel organization the connection between the device and the controller is used to transfer both commands and data.

High-end Personal Computers and workstations usually have a bus and controllers attached to the bus(see Figure 4.2). Each controller controls a number of devices. In most cases the controller controls one or more identical devices. In other cases the controller might control a variety of different devices.

The existence of controllers is generally transparent to the end user but very important for the designer of the device drivers. All the operations of devices that share one controller are controlled from the same set of registers. In addition in a lot of cases the controller limits the number of operations that can be submitted simultaneously to devices that share the same controller. Finally, devices that share the same controller usually share the same interrupts to the system.

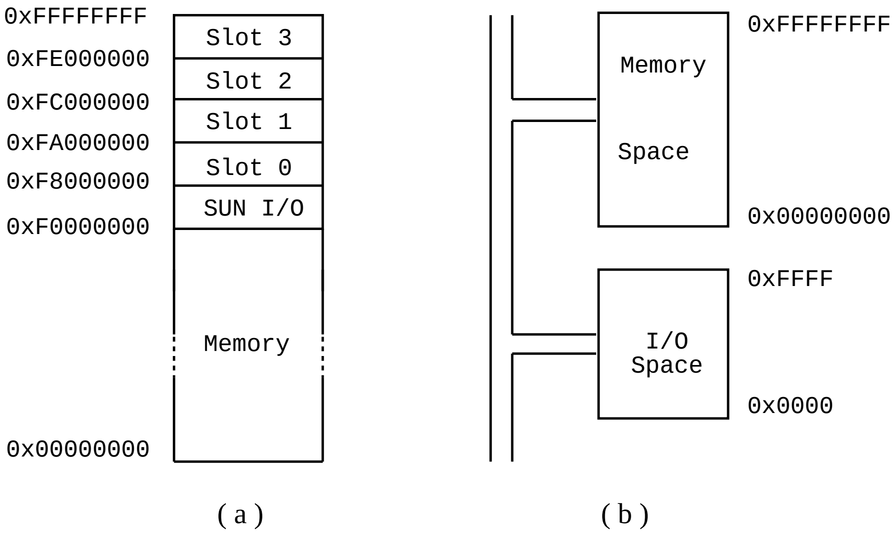


Figure 4.3: (a) The Sun's SBus memory-mapped architecture. (b) The IBM's Micro Channel I/O architecture uses a 16-bit I/O address space.

## 4.2 Addressing an I/O Controller

The system communicates with a device in order to transfer data and to set control parameters. For example, every serial line allows the baud rate to be set by software. In more advanced machines, mainly mainframes, controllers can even execute I/O programs found in memory.

The controller of a device can be addressed by writing or reading bytes, words or long words at specific addresses in an address space. Each of these addresses corresponds to a control or data port of the controller. Depending on the architecture there are two ways a program can access the ports of a controller. These two ways are described in the next two subsections.

### 4.2.1 Memory-Mapped I/O Address Space

If physical memory-mapped addressing is used, the ports of the controller are mapped to a range of addresses in the physical memory of the computer. When a program reads from or writes to these addresses the ports of the controller are accessed. The documentation of the controller states which addresses correspond to which port. This approach is called **memory-mapped** Input/Output and is supported by nearly every general purpose processor. It has the advantage of simplifying the instruction set of the processor since no special purpose Input/Output related instructions are needed. On the other hand it partitions the physical memory address space since certain ranges are allocated to Input/Output controllers. In addition, memory-mapped I/O complicates the data cache management[49]. As an example of memory-mapped architecture, Figure 4.3.(a) shows the memory map of systems using the SBus[19]. The SBus is a bus used in the Sun SPARCStation series of computers. One address space is used for both the memory and the I/O devices.

### 4.2.2 Separate I/O Address Space

Some processors provide a relatively small address space dedicated to Input/Output. This address space is not accessed by the normal `load` and `store` instructions of the architecture but by special `in` and `out` instructions. The use of a special I/O address space makes the processor instruction set more complicated. In addition it makes the access of ports more difficult since usually very few addressing modes are provided with the I/O instructions. In processors with several protection levels the I/O instructions are protected instructions. Therefore, a special mechanism is needed for applications that need exclusive access to a device and want to avoid trapping in the kernel. For instance, the Intel386 processor[29], associates an I/O bitmap with each task. Each bit in the bitmap represents an address at the I/O address space. A task can access an I/O address only if the corresponding bit in the bitmap is reset. (In memory-mapped I/O architectures the common memory management techniques can map a segment of memory to the virtual address space of a process.) Figure 4.3.(b) shows the approach taken in the IBM PS/2 computer[26] which uses the Micro Channel architecture. Micro Channel provides two separate address spaces. The one is allocated to Random Access Memory. The other is an I/O address space used by I/O controllers.

Accessing a port of a controller from a high-level language, like C or C++, in a memory-mapped I/O architectures is relatively easier than accessing a port in an architecture that uses a dedicated I/O address space. In the former case a variable bound to the address of the port can be used to access I/O ports. Assigning to the variable is equivalent to writing to the port. Reading the contents of a variable is equivalent to reading form the port. In architectures with separate I/O address space a special inline function written in assembly is needed since high level languages usually do not have the concept of I/O instructions.

## 4.3 Synchronization Among I/O Controllers and the System

For most devices a CPU initiates an I/O operation through a controller. Then the controller executes the operation and completes after some time. The CPU has to know when the operation is completed in order to resume processes waiting for this completion. There are two basic approaches used to synchronize a CPU and an I/O controller. The CPU can test the status of an I/O port to determine whether I/O is taking place. The device is responsible for setting the status of this port upon completion of the operation. This kind of synchronous Input/Output is traditionally called **I/O polling**.

However polling is very wasteful for systems that support multiple processes. While one process is doing I/O the CPU could schedule another process instead of "busy-waiting". In addition, for some asynchronous devices like a serial line, input comes asynchronously. The CPU would waste a lot of cycles checking for input from such asynchronous devices.

Most modern architectures provide an interrupt mechanism that overcomes the problems mentioned above. Usually, the interrupt scheme is very closely related to the exception scheme of the processor. Each controller is associated with an interrupt line that connects, possibly indirectly, the controller with a CPU. When special events occur, like the completion of an operation or an error condition, the controller notifies the CPU by raising the signal on this line. Depending on the architecture one or more devices might share the same interrupt line. It is very common for devices that share the same controller to share the same interrupt line as well. In most architectures interrupt lines reach the CPU through a special interrupt controller. The CPU can disable and enable certain interrupts by programming the interrupt controller.

In all the modern processors Input/Output interrupts are treated as special exception cases. When an exception occurs a number of steps are taken to figure out the software routine to be invoked. This software module is usually called **exception handler**. The process of invoking the right exception handler is performed by a combination of software and hardware. Most CPUs[38, 29, 44] allow up to 255 different types of exceptions. A small number—usually between 8 and 16—of these exceptions are assigned to the Input/Output subsystem. When an exception takes place the normal flow of control is interrupted and an exception handler is invoked. In architectures that support **auto vectoring** the CPU keeps a pointer to a table in the memory where the addresses of the exception handlers are kept. Upon an exception

| *Bus Name* | IEEE standard | *Address Width* | *Data Width* | *Peak Performance* |
|---|---|---|---|---|
| IEEE VMEBus | P1014 | 16,32 | 16,32 | 40 Mbytes/sec |
| IEEE NuBus | P1196 | 32 | 32 | 37.5 Mbytes/sec |
| Motorola Multibus II | P1296 | 32 | 16,32 | 40 Mbytes/sec |
| IBM Micro Channel | - | 16,24,32 | 8,16,32 | 17 Mbytes/sec |
| Sun SBus.1 | - | 32 | 16,32 | 66 Mbytes/sec |

Table 4.1: A number of 32-bit system buses and some of their characteristics.

the proper handler is called by the hardware. In other architectures[33] one general exception handler finds and calls the proper exception handler.

The use of interrupts enhances the performance of the system. However, the programmer should be very careful because of the asynchronous nature of the exceptions. When an exception occurs part of the context must be saved. In addition data that is supposed to be accessed by both an interrupt handler and another process must be protected inside a critical region. In a device that supports a lot of interrupting devices the interrupts must be prioritized. Devices like timers, keyboards and network controllers that need immediate response must be given higher priorities. While an interrupt from a device is served interrupts from devices in the same or lower priorities must be disabled.

## 4.4 Programmed and DMA Data Transfer

Data is transferred from the I/O controllers to the system memory over an interconnection network that connects the memory and the device. The most common way to interconnect the memory and a controller is through a bus. A bus is a collection of data, address and control signals and a number of protocols that explain how these lines will be used by the users of the bus. Some buses along with their characteristics are shown in Table 4.1 (source [6]).

Table 4.1 shows that the largest quantity of data that can be transferred over a bus is limited to a number of bits. Therefore, transferring a block from a disk or transmitting a packet over the network must be split into a number of word transfers. The transfers can be accomplished by either a CPU or by a special DMA processor. DMA processors are specialized processors[25] which can transfer data directly from a controller to memory over the bus. The first type of Input/Output is called **Programmed I/O** while the second one is called **Direct**

**MemoryAccess I/O**. CPUs and DMA processors might have to use the same bus during the DMA operation. This is called **cycle-stealing**.

Another performance problem is introduced with devices that transfer large amounts of data in one operation. For instance, an ethernet controller should transfer a block of up to 1400 bytes and a disk controller might want to transfer a number of contiguous blocks to the memory at once. However the target or destination memory might not be contiguous. There are two reasons why the source or target memory might not be contiguous. The first reason is that a contiguous region in a virtual memory space is not necessarily contiguous in the physical memory address space. The second reason is that devices like the ethernet support variable length packets consisting of a number of header and trailers. Using a CPU to copying all these headers and trailers from discontiguous memory to a contiguous region is time consuming.

Some architectures support transfer to discontiguous areas in the physical memory using one or both of the following mechanisms. The one is called Direct Virtual Memory Access or DVMA. In DVMA, the addresses generated by the DMA controller are first translated by a Memory Management Unit. The CPU might use the same MMU to access the address space of a process. The second mechanism provides a so called **scatter-gather** DMA controller. A scatter-gather DMA controller is basically a smart DMA controller which takes as input the description of the discontiguous memory and then transfers the device data. The two techniques can be combined together as is the case with the Sun's ethernet controller on the SPARCStation.

Using DMA Input/Output and interrupts to notify the completion of an operation is the fastest way to transfer data from a controller to the memory. However, the DMA processor itself along with the mechanism of cycle stealing is quite expensive and is avoided in cheap microcomputers. For example in the Macintosh series of computers only the latest and most expensive model—Macintosh II fx—uses DMA transfers for the standard controllers.

In systems that support DMA transfers extra care should be taken if virtual memory is also used. Usually, in such systems a user specifies an address in his virtual address space. However, the DMA processors are programmed with physical addresses. It is the responsibility of the software to translate the virtual address of a buffer to the physical address before programming the DMA.

A second problem appears in systems that support paging or swapping of virtual address spaces. While data is transferred from a device to a frame in the physical memory the page must be in the physical memory of the computer. Special operating system routines should be used to "lock" the page in the physical memory while I/O is taking place.

## 4.5 Summary

This section described the most widely used I/O architectures. The I/O architectures form the problem domain of a device management framework for a general purpose operating system. As I discussed in section 2.3.2 the problem domain of an object-oriented framework should be very well understood should a framework have any chances of being useful. Controllers and devices are key abstractions in this domain. The devices provide the functionality, however, devices are accessible via controllers. Each controller directly or indirectly "controls" a number of devices. In memory-mapped architectures controllers occupy a part of the processor's physical address space. Other architectures use a dedicated I/O address space. Devices connected to the same controller usually use the same interrupt line to the processor. The next chapter describes how these observations of the problem domain were used to design a device management framework for the *Choices* operating system.

# Chapter 5

# *Choices* Device Management

*Choices* device drivers consist of objects of two classes, *DevicesControllers* and *Devices*. Each *DevicesController* represents a physical or logical controller that controls a number of *Devices*. *Devices* are accessible to the user through the *Choices* kernel `NameServer` object. In contrast to other operating systems, like UNIX[3] and V[9], *Choices Devices* usually do not provide the topmost I/O interface to the programmer. They simply provide the minimum interface to the hardware that will not sacrifice efficiency for modularity. The *Choices* uniform I/O interface is provided by higher level abstractions like *MemoryObjects*, *InputStreams* and *OutputStreams*. *Devices* can be **converted** to these higher level abstractions invoking the method `asA()`.

The *Choices* drivers are highly influenced by the UNIX Operating system device drivers[3, 34, 16, 28]. However, there are several important differences. The most important difference is the object-oriented design and implementation that makes the *Choices* approach much cleaner, understandable and reusable. The second difference, is that the *Choices* design of the I/O subsystem leads to **lighter** device drivers than the traditional UNIX drivers. This means that the drivers do the minimum machine dependent operations. Finally, *Choices* does not provide naming for the *Devices* through the file system. Devices are not special types of files and so they are not treated as special kinds of files. Naming for the *Choices* devices is provided through the kernel name server.

*Choices* machine independent code provides the classes *DevicesController* and *Device* to be reused for device drivers implementation. A number of abstract subclasses have also been implemented for several well known and understood types of devices. This hierarchy is shown

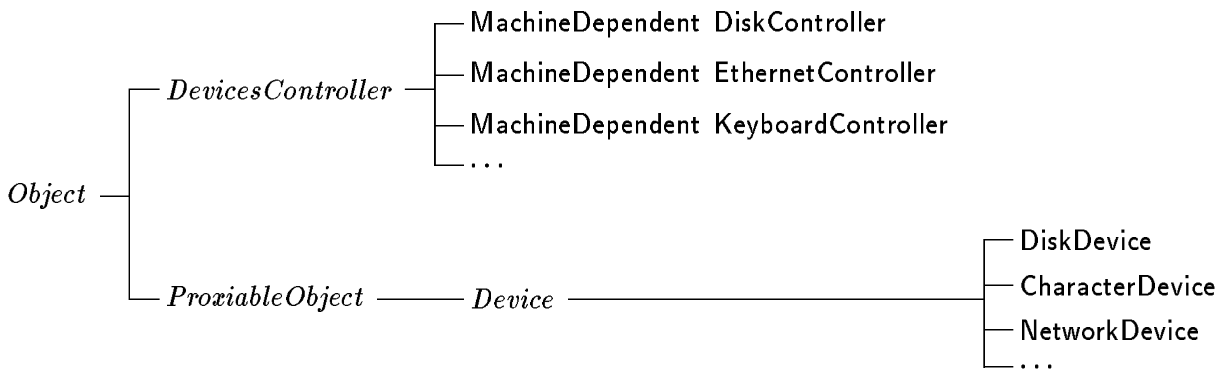


Figure 5.1: A part of the extensible *DevicesController* hierarchy.

in Figure 5.1. The next section describes the responsibility and the interface of the *DevicesController* class. Section two describes the class *Device*. Finally section three describes how the device drivers are configured and linked with the rest of the system.

## 5.1 The *DevicesController* Class

Each *DevicesController* represents a physical or logical controller of the computer system that controls a number of devices. Each device inside the controller is represented by a `DeviceId` in the range `[0..MaxDevices]`. The id `0` is reserved for the controller itself. For instance in the IBM PS/2 implementation of *Choices* the class PS2DisketteController is a class representing the controller of the diskette drives. The controller is located on the motherboard and has up to two PS2DisketteDevices. The first one has id 1 and the second has id 2. The PS2DisketteController is addressed by the id 0.

A *DevicesController* performs a number of operations on behalf of its *Devices*. *Devices* are the only clients of a *DevicesController*. A *DevicesController* is not visible outside the device management framework. Services to other frameworks are provided by the *Device* class. Table 5.1 shows the protocol of the class *DevicesController*. The *DevicesController* class takes care of administrative work for the controller and the devices. Input and Output for the devices of the controller is handled by the subclasses of this class.

| *DevicesController* | *Object* | |
|---|---|---|
| | `DevicesController` | `( int numOfDevs )` |
| | `~DevicesController` | `()` |
| `IOStatus` | `initialize` | `( slot, address, offset, exceptions, flags)` |
| `Device *` | `attach` | `( DeviceId, flags, IOStatus & )` |
| `IOStatus` | `detach` | `( DeviceId, flags )` |
| `IOStatus` | `sendCommand` | `( DeviceId, Command * )` |
| `int` | `interruptHandle` | `( char * info ) = 0` |

Table 5.1: The protocol of the *DevicesController* class.

### 5.1.1 Construction of a *DevicesController*

Each *DevicesController* keeps a list of all its *Devices* that represent physical devices connected to the system. During the construction sequence, the constructor of the *DevicesController* sets all the devices to detached. I describe later how the devices are attached by calling the method **`attach()`**. Constructing a *DevicesController* does not mean that the controller is present or that the hardware functions properly. The **`initialize()`** method, described below, is responsible for testing the presence and the proper function of the controller and should be called immediately after the construction of the object. So, the constructor of a typical subclass has an empty body.

### 5.1.2 Initialization of a *DevicesController*

Each *DevicesController* usually represents a physical controller of the computer. The controller can be either on the motherboard or in an expansion slot in the backplane of the machine. Architectures usually allow more than one controller of the same type to be mounted in the backplane of the machine. In this case each card can be programmed[1] to be accessed at different addresses and to generate different interrupts. This information is passed to the object representing the controller when it is initialized.

A *DevicesController* should be initialized by calling the virtual method **`initialize()`**. This method is responsible for:

1. mapping the memory of the controller into the virtual address space of the kernel.

[1] Older cards are programmed manually by setting jumpers or switches. Other cards read the configuration from registers in the motherboard or are geographically addressed.

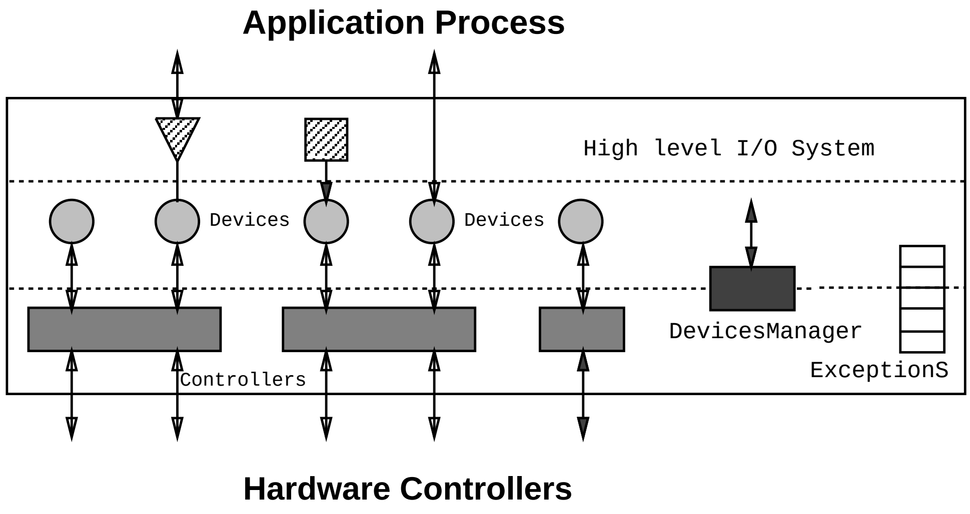


Figure 5.2: *DevicesControllers* are at the bottom of the kernel. They interact only with their *Devices*, the DevicesManager and the interrupt handling subsystem.

2. building and setting the exceptions generated by the controller.

3. finding the physical controller and making sure that it operates properly.

The first two items above are handled by the `DevicesController::initialize()`. The third item is handled by the controller-dependent `initialize()`. The controller-dependent `initialize()` is expected to call `DevicesController::initialize()` to perform items 1 and 2, and then perform item number 3. The destructor of the *DevicesController* is responsible for deleting the exceptions and unmapping the memory.

As shown in Table 5.1, the method `initialize()` takes five arguments:

- `slot`: This argument is for *DevicesControllers* representing controllers on cards mounted in the expansion slots of the machine. It shows in which slot the card is expected to be found.

- `exceptions`: This is a sequence of exception numbers that will be generated by interrupts from this device. For instance, a legal actual parameter for this argument would be `{ "18", "21" }`. This means that if the exceptions 18 or 21 are ever generated this object should be notified. Each controller usually generates only one exception. This structure gives flexibility for controllers that generate more than one exception.

- **address, offset**: These parameters give a description of a region where the ports of the card are expected to be found in a memory mapped architecture(see section 4.2). The method `DevicesDriver::initialize()` is responsible for mapping the region one to one in the kernel address space.

- **flags**: The parameters described above are useful in the case where more than one controller of the same type are present in a machine. In most other cases this information is not needed and the driver can autoconfigure itself. For example, in most modern architectures it is possible to find what type of card exists in each slot. In the PS/2 architecture this can be done by reading the Programmable Option Select registers[26]. In the Sun's SBus this can be done by interpreting Forth programs, expected to be found at a specific address for each slot[19]. If the `AUTOCONFIGURATION` bit is set in the flags parameter the *DevicesController* ignores the parameters and autoconfigures itself. Autoconfiguration should be done by the controller dependent `initialize()` which then calls the `DevicesController::initialize()`. Other bits in the flags argument are interpreted by the controller-dependent `initialize()`. In case of autoconfiguration the *DevicesController* is also responsible to find which physical devices are connected to the physical controller it represents. For each connected device it informs the DevicesManager of the system(see sectio sectionn 5.3). The DevicesManager calls back the *DevicesController* to build the *Device*, which is then bound to the name server.

The `initialize()` method returns a status to the caller. If the *DevicesController* finds and properly initializes the hardware controller, the status will be `IOSucceeded`. Otherwise, an error will be returned to the caller. If the initialization fails, the caller should delete the *DevicesController* object. Figure 5.3 shows a outline for the method `initialize()`. In section 5.3 I discuss how this method is called by the *Choices* device drivers installation mechanism.

### 5.1.3 Attaching and Detaching *Devices*

The methods `attach()` and `detach()` are used to get and release *Devices* from a controller. The `attach()` method takes a `DeviceId` as an argument and returns a *Device* upon successful completion. It ensures that the devices that are bound to the name server are present and operate properly(see section 5.3). The first time the method `attach()` is called the hardware is

```
IOStatus
DevicesController::initialize( ... ){

     /*** 1 Set the exceptions */
     for( each exceptionNumber supported by the controller ){
        exc = new DevicesControllerException( this );
        thisCPU()->ExceptionManager()->addException( exceptionNumber, exc );
     }

     /*** 2 Set the memorymap */
      if( address != 0 ){
         mo = new PhysicalResisdentMemoryObject( address, offset, ... );
         theKernel->domain()->add( mo, ... );
      }

      return( IOSucceeded );
}

/******************************************************************/

IOStatus
<Concrete subclass of DevicesController>::initialize( ... ){

     /***1 Do autoconfiguration */
     if( flags & AUTOCONFIGURATION ){
        < Hardware dependent >
        < set the exceptions and region >
     }

     /***2 Call the initialize of the root class */
     ios = DevicesController::initialize( ... );
     if( ios != IOSucceeded ) return( ios );

     /***3 Reset the controller */
     < reset the controller >

    if( flags & AUTOCONFIGURATION ){
          < For each physically connected device
              inform the DevicesManager>
     }

     return( IOSucceeded );
}
```

Figure 5.3: An outline of the **initialize()** method.

```
Device *
DevicesController::attach( DeviceId did, int flags, IOStatus & status ){

     if( _devices[ did ] != 0 )
          _devices[ did ]->reference();
     return( _devices[ did ] )
}

/**********************************************************************/

Device *
<Concrete subclass>::attach( DeviceId did, flags, IOStatus & status ){

     if( !this->isPresent( did ) ){
           <controller dependent checks>
          _devices[ did ] = new <Device>( this, did, .... );
          this->setPresent( did );
     }

     Device * d = DevicesController::attach( did, status );
     if( d == NULL ) status = IOOutOfMemory;

     return( d );

}
```

Figure 5.4: A skeleton for `DevicesController::attach()` and a concrete subclass

checked to make sure that the physical device is present and a new *Device* object is constructed and returned. Subsequent calls for attaching the same device will return the same object. If for any reason the *Device* cannot be constructed a `NULL` pointer is returned and the argument `status` indicates the reason. The controller dependent `attach()` checks if the device has already been attached. If it has not been attached it resets the device and checks if it is properly installed. Then it calls the `DevicesController::attach()` to build and return the device. Figure 5.4 shows the outline of the attach methods for the `DevicesDriver` and for a concrete subclass.

A device can be detached by calling the method `detach()`. The reference count of the *Device* should be 1, that is the Device should not be used by any process, otherwise the `detach()` will fail. However, if the flags bit `NODELAY` is set the device will be detached despite the number of references it has. Other flag bits are controller dependent.

### 5.1.4 Sending Commands to a Controller

The most important method of a *DevicesController* object is the `sendCommand()` method which sends commands to the driver. The commands can initiate Input/Output, perform control functions or extract information about the controller and its devices. This method takes two arguments. The first argument is the `DeviceId` of the device to which the command is sent. The second argument is a *Command* object, described later. The *DevicesController* interprets the command and returns a status. If the command succeeds the status will be `IOSucceeded`. Otherwise the error is specified.

The interface between the *DevicesControllers* and the *Device* is based on the `sendCommand()` method for three reasons. The first reason is to make easier the interconnection of *Devices* and *DevicesControllers*. A *Device* can send a command to any *DevicesController*. If the *DevicesController* understands the command it will perform the operation. If a strong interface[2] had been adopted DiskDevices would have been able to communicate only with DiskControllers. However, other devices can be connected to a hierarchy of controllers. The controllers in the hierarchy may often simply establish a channel and pass the commands to the appropriate controller. For example, this is the case with the IBM mainframes[21, 25] where a disk is connected to storage

[2] I use the term strong interface to refer to a type checked interface and the term weak interface to refer to a message based interface.

```
typedef unsigned long CommandType;

const CommandType UndefinedCommand = 0;

class Command {
public:
    Command( type );
    CommandType type();

protected:
    CommandType type;
};

inline Command::Command( CommandType t = UndefinedCommand ){ type = t; }
inline Command::type(){ return type; }
```

Figure 5.5: The lightweight class *Command*.

directors and channels. The second reason for not adopting a strong interface is to make the system more flexible. The protocol of a *DevicesController* can be extended without recompiling existing *Devices* using this controller. Finally, the third reason is that in some cases requests to a *Device* have to be converted to *Command* objects in order to be queued. Later, these commands can be directly forwarded to the *DevicesController*.

### 5.1.5 The *Command* Class

*Commands* are data structures that contain the type and the data of the requested function. Each *Command* has a `type` field inherited from the *Command* superclass. All instances of the same subclass of *Command* have the same type. Types should be unique system-wide. This allows a *DevicesController* to interpret the contents of a *Command* object only from the value of the type field.

Some common types are predefined. A new *DevicesController* should reuse old *Commands* as much as possible. New *Commands* should be added only if it is necessary. For instance, the command SetBaudRateCommand is used to set the baud rate of a serial line. A system programmer who designs a driver for a new serial line controller is supposed to reuse this command. This way commands will be the same across different controllers that provide the same or similar functionality.

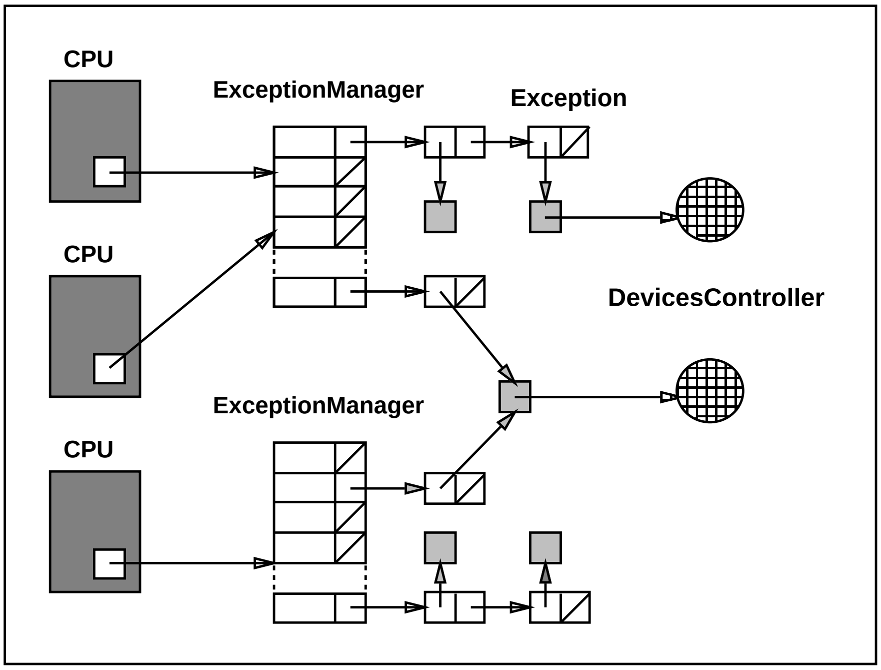


Figure 5.6: The logical structure of the objects participating in the exception management in *Choices*.

### 5.1.6 Interrupt Handling

In *Choices* each hardware exception is associated with a list of *Exception* objects as shown in Figure 5.6. When an exception is raised in a *CPU* the list of *Exception* objects corresponding to this exception in the *ExceptionManager* of this *CPU* is traversed and the message `raise()` is sent to each *Exception* object until an object handles the exception.

The `Exception::raise()` method returns a Boolean value. A value of `True` means that the *Exception* was responsible and handled the interrupt. A value of `False` indicates that the *Exception* was not responsible for the generated exception. The *CPU* then continues with the next *Exception*. If no *Exception* claims responsibility for the interrupt a diagnostic is logged and the interrupt is ignored.

When a *DevicesController* is initialized(see Figure 5.3) it adds DevicesControllerExceptions into the exception manager of the *CPU* to handle the hardware exceptions. Each DevicesControllerException has a reference to a *DevicesController* object. The receipt of the `raise()` message leads to the invocation of the virtual method `interruptHandle()` on the *DevicesController*.

| *Device* | *ProxiableObject* | |
|---|---|---|
| | Device | () |
| | ~Device | ( DevicesController *, DeviceId ) |
| ProxiableObject | asA | ( Class *, flags ) |
| IOStatus | sendCommand | ( Command * ) |
| IOStatus | getCommand | ( Command * ) = 0 |
| void | commandCompleted | ( Command * ) = 0 |

Table 5.2: The protocol of the *Device* class.

A *DevicesController* gets this message when the controller has issued an interrupt. The *DevicesController* determines the cause of the interrupt and takes the appropriate actions.

## 5.2 The *Device* Class

*Devices* in *Choices* represent hardware devices of the computer system attached to physical controllers. For instance, a serial line attached to a serial line controller is an example of a *Choices Device*. *DevicesControllers* are hidden at the bottom of the kernel. As depicted in Figure 5.2, *DevicesControllers* act only as servers of their *Devices*.

In this framework *Devices* have three responsibilities:

- The first responsibility is to form a connection between the device drivers and the higher level I/O subsystem.
- The second is to encapsulate all the information related to the devices they represent. *DevicesControllers* keep their *Devices* as part of their state and use this information when needed. For instance, a *Device* representing a disk knows the layout of the disk and the current position of the heads. A device representing a serial line knows the baud rate and other parameters of the line.
- Finally, the third responsibility is to provide protection, exclusive access—if needed—and reference counting for the devices they represent.

The class *Device* is a subclass of the class *ProxiableObject*(see Table 5.2). *ProxiableObject* is a *Choices* abstract class that provides reference counting, conversion and user level access facilities. The *Devices* protocol is described in the next subsections.

### 5.2.1 The Functionality of a Device

A *Device* acts as server for applications and other parts of the *Choices* kernel, like the file systems. Each subclass of the *Device* class defines a number of methods which form the protocol of the *Device*. In most cases, the methods simply build an appropriate command and forward the command to the corresponding *DevicesController* by invoking the `sendCommand()` method. However, different approaches can also be taken. For example, if a *Device* that represents a disk knows the size of the disk, a request for the size of the disk can be handled locally. Input/Output commands are always forwarded to the responsible controller unless caching is taking place.

### 5.2.2 The *Choices* Conversion Mechanism

One very important mechanism provided to *Choices* objects that they inherit from the class *ProxiableObject* is that they can be **converted** to objects of other classes. The term conversion comes from the Smalltalk[20] programming language. In Smalltalk, conversion mechanisms are used by the *Collection* classes. For instance, invoking the **asBag** method on a *Collection* object returns a new object. The new object will be an instance of the class Bag and its contents will be based on the original object. Conversion does not modify the original object. Instead it builds a new object "based" on the original object.

In *Choices* we introduced a conversion mechanism[37] that is an extension of Smalltalk's mechanism. Every *ProxiableObject* in *Choices* responds to the virtual method `asA()`. This method takes two arguments. The first argument is a Class object. The second argument is a long integer whose interpretation depends on the objects participating in the conversion. The method returns a new object that is an instance of the class passed as the first argument or an instance of any of its descendants.

In *Choices* conversion works by means of double dispatching[22]. Each class in *Choices* is reified as a first class object. *Choices* objects that represent static compile-time C++ classes are instances of the class Class[36][3]. The method `Class::classSupports()` takes an object as an argument and returns a `Boolean` value indicating whether it **supports** this object. A

[3] In this thesis when I say class I mean the C++ compile time entity. When I say a Class I mean a run-time object that represents a C++ class.

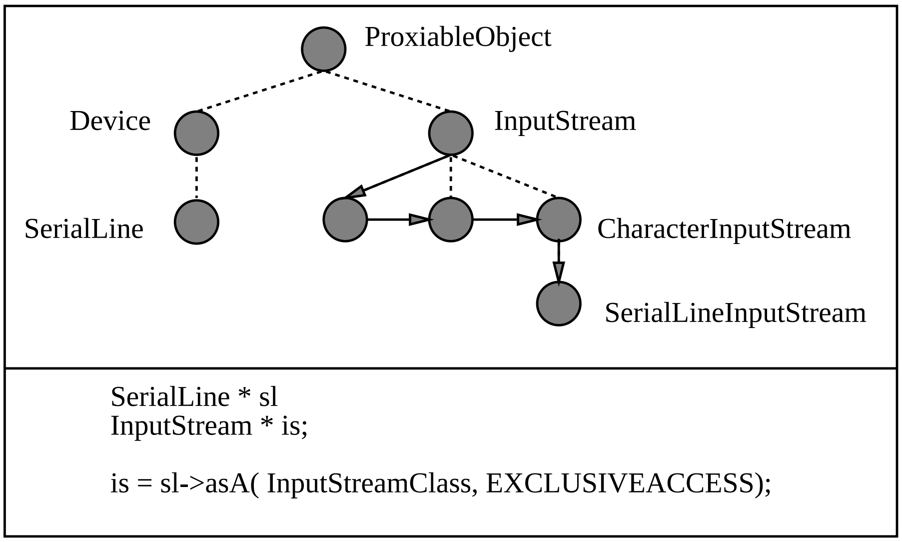


Figure 5.7: A SerialLine is converted to an *InputStream*. The double dispatching mechanism finds that the SerialLineInputStream class is the *InputStream* that supports SerialLines.

class supports an object if the constructor of the class can take this object and an integer as arguments and build a new object.

When the method `asA( Class *, int )` is sent to an object, the object forwards the method to the Class passed as argument. As depicted in Figure 5.7, the Class then traverses the tree of Classes of which it is a root until it finds a descendant class that responds positively to the message `classSupports()`[4]. Then this Class is used to construct the new object.

Double dispatching has the advantage that a class `foo` can be written without knowing which classes, in the future, will support `foo`. This is very well suited to open systems. However, it has the disadvantage that conversion is slower since a tree of classes should be traversed to find the exact subclass that supports this object. An implementation based on caching of earlier results can enhance the performance.

### 5.2.3 Conversion of *Devices*

The conversion mechanism is used to establish a connection with a device. When a user wants to get a device he first has to look up the name of the device in the kernel's name server. *Devices* are bound to the name server by the devices configuration mechanism(see section 5.3).

[4] I assume here that only one class in this hierarchy supports the object

The method `NameServer::lookup()` takes three arguments. The first is the symbolic name of the object, the second is the class to which the object should be converted and the third one is a long integer. In the case of a *Device* the long integer is the flags. If the name server finds an object under this name then it invokes the method `asA()` on it, passing the second and third argument of the `lookup()` method as parameters. The protocol of the method `asA()` is inherited by the class *ProxiableObject*, as described in the previous section. The method `NameServer::lookup()` returns to the user the object returned by the method `asA()`. If any error occurs in this process, a value of zero is returned.

The method `Device::asA()` first checks the flags to find if exclusive access is requested. If exclusive access is requested, then the *Device* checks if such permission can be given and acts accordingly. Then the `ProxiableObject::asA()` is called which uses the *Choices* double dispatching mechanism to get an instance of the given class or any of its descendants—based on the *Device*. This is how serial lines are converted to *InputStreams* and *OutputStreams* and disks are converted to *MemoryObjects*.

As I mentioned earlier, the use of double dispatching is very important because it frees the *Device* from having to know the open collection of classes that support this *Device*. For instance, the class DiskMemoryObject is a *MemoryObject* that performs reads and writes using a DiskDevice. The class DiskMemoryObject supports DiskDevices. However, the class DiskDevice is not required to be aware of it. New classes can be added to the system that also support DiskDevices but the DiskDevice will not have to be modified.

## 5.3 Configuration of Device Drivers

The concrete class DevicesManager is responsible for configuring the *Choices* device drivers. The *DevicesController* can be linked either statically or dynamically to the kernel. Each *DevicesController* should inform the DevicesManager that it was loaded by passing its name and its constructor.

The DevicesManager keeps as part of its class state a "map" of all the drivers, hardware controllers and hardware devices of the system. This information is provided by either the system administrator or, in case of systems that support automatic configuration, by a machine dependent procedure. For each hardware controller the following information is kept:

| DevicesManager | *Object* | |
|---|---|---|
| void | addDriver | ( DriverName, constructor ); |
| void | addController | ( CtrlName, drvrName, address, offset, slot, vectors, exceptions, flags ); |
| void | addDevice | ( CtrlName, deviceName, DeviceId ) |
| void | removeDriver | ( DriverName ); |
| void | removeController | ( CtrlName ); |
| void | removeDevice | ( CtrlName, DeviceId ) |

Table 5.3: The protocol of the DevicesManager class.

- The name of the *DevicesController* that will act as driver of this controller.
- The slot in which this controller is mounted, if the controller is not on the motherboard.
- A list of exception numbers that are generated by this controller.
- A description of the address space that the controller occupies.
- The devices that are attached to this controller and the name under which they should be bound to the name server.

When a new *DevicesController* is loaded the map of the system is checked for any controllers that have declared the particular *DevicesController* as their driver. For each of these controllers a *DevicesController* is built and initialized. Then the designated devices are **`attached()`** and bound to the name server. An outline of the **`addDriver()`** method is shown in Figure 5.8.

Controllers and devices can be dynamically added and removed from the system by calling the method **`addController()`** of the DevicesManager class. This method looks up a loaded *DevicesController* that can take care of the controller. If such a class exists an instance is built and initialized. For every device attached to this controller, the method **`attach()`** is called and the returned *Device* is bound to the name server. If a *DevicesController* with such a name has not been loaded to the system the entry is saved for later use. A similar procedure is followed for removing a controller or device from the system.

## 5.4 Summary

This chapter has discussed the two hierarchies of the Device Management framework. The classes at the top of these hierarchies are reused to build *Choices* device drivers. The class

```
IOStatus
DevicesManager::addDriver( name, install ){

    /*** 1 store the name and the install function for possible later use */

    /*** 2          */
    for every controller in the system
       if the controller is supposed to be taken care by this DevicesController{
            DevicesController * dc = (*install)();
            dc->initialize( ... );
            if( error ){
                log the error;
                delete dc;
                continue;
            }/* if */

            for any attached device to this controller{
                 Device * dev = dc->attach( ... );
                 if( dev != NULL )
                      TheKernel->nameServer()->bind( dev, ... );
                 else
                      log the error
            }/* for */
        }/* if */

}
```

Figure 5.8: An outline of the `DevicesManager::addDriver()` method.

```
FiSh> ns Device
>kbd          CharacterDevice[0x2634c0]{4}
>fd0          DetailedDiskDevice[0x270440]{2}
>tty0         SerialLineDevice[0x2635e0]{4}
>rd9          ContiguousDiskDevice[0x263600]{2}
>hd0          BufferingDiskDevice[0x484000]{3}
>hd1          BufferingDiskDevice[0x508000]{3}
>dis0         CharacterDisplayDevice[0x2631e0]{2}
>dis1         CharacterDisplayDevice[0x263200]{3}
```

Figure 5.9: An example of *Devices* existing in the *Choices* kernel name server on the PS/2 computer.

*DevicesController* represents hardware controllers. Subclasses of this class encapsulate the functionality of machine dependent controllers. *DevicesControllers* act as servers for *Devices*. A *Device* encapsulates the state and provides the functionality of a hardware device. *Devices* are bound to the name server where they can be looked up by applications and by other parts of the kernel. The DevicesManager class keeps a map of all the drivers, controllers and devices in the system.

*Devices* are converted to other objects by using the *Choices* conversion mechanism. Double dispatching allows new classes that support a particular kind of *Device* to be loaded and instantiated without modifying the *Device*. The open interface between a *DevicesController* and a *Device* makes it possible to plug together various different devices and controllers.

The device management framework has been implemented on IBM PS/2 computer. The **FiSh**[5] command **ns** takes one argument, a string, which is the name of a class. It shows all the objects in the NameServer that are instances of this class or any of its descendants. The first column shows the names of the objects. For each object the second column shows the class of the object. Figure 5.9 shows an example of *Devices* that exist in the NameServer in the PS/2 implementation of *Choices*.

The next two chapters describe two subclasses of the *Device* class. The first is the DiskDevice class. Objects of this class are expected to represent disk devices. The second is the CharacterDevice which is expected to represent devices like serial lines and keyboards.

[5] **FiSh** is the most widely used shell in *Choices*.

# Chapter 6

# A Reusable Disk Driver

This chapter describes the design of a subframework of the device management framework. A programmer can reuse this subframework to implement disk drivers. The framework is based on the class DiskDevice which is a *Device* representing disks. The DiskDevice is usually a client of a controller dependent *DevicesController* class representing a specific disk controller. However, the DiskDevice can be the client of other classes as well. For instance, the DiskDevice can be used as a *Device* of a *SCSIController*, a controller controlling a Small Computer System Interface bus[11]. Furthermore, a DiskDevice representing a RAM disk does not need a *DevicesController* at all.

The classes presented here do not expedite the implementation of a device driver for any arbitrary disk. However the framework is general enough to cover several cases of different disks and disk controllers. The first section of this chapter describes how controller dependent classes representing disk controllers should behave. The second section describes the class DiskDevice. The third section gives a detailed example of how the framework works. Finally the last section, discusses the interface of the driver with the rest of the *Choices* kernel.

## 6.1 The Machine Dependent Disk Controller Class

This section discusses how machine dependent *DevicesControllers* that represent disk controllers should behave. In the rest of this chapter the class DiskController stands for a machine dependent *DevicesController* representing a disk controller. A DiskController is a *DevicesControllers*, so it is not a direct client of an Input/Output transfer originated outside of the framework. Instead, DiskDevices are responsible for sending commands to the DiskController. This approach has three advantages.

The first is that the DiskDevice can translate high level operations to primitive commands that the controller understands. The implementor of the DiskController class can concentrate on the details of the I/O controller. The second advantage of this approach is that arguments passed by malicious or careless users will be checked by the DiskDevice before being passed to the controller class. Finally, the third and most important advantage is that the DiskDevice can be reused by designers of different controllers and devices.

The controller dependent class has three responsibilities. The first responsibility is to receive *DiskCommands* from the class DiskDevice and execute them. The second responsibility is to inform the DiskDevice when an operation is completed.

The third responsibility of the DiskController is to know if the controller is busy doing Input/Output. If the controller is not busy then an interrupt from a device for a seek completion causes the driver to get new Input/Output commands from the DiskDevices and start executing them. DiskDevices are responsible for knowing the status of the devices they represent.

### 6.1.1 Sending Commands to a Disk Controller Class

The DiskController should understand *Commands* that are subclasses of the class *DiskCommand*. This class and some of its subclasses are shown in Figure 6.1. The fields of the class have the following meaning:

- `_devId` is the `DeviceId` of the device that participates in the transfer.
- `_operation` specifies the kind of transfer, that is if it is a disk read or write.
- `_startingBlock` is the first block to be transferred. Concrete subclasses of this class will specify how many blocks will be transferred and where in the memory they will be transferred.
- `_nextCommand, _lastCommand` These are fields used by higher layers to link *DiskCommand* objects together and should be ignored by the DiskController.
- The field `_retries` specifies how many times this command should be retried before reporting an error.

```
class DiskCommand : public Command{
public:
      DeviceId        _devId;              /* for which device */
      DiskOperation   _operation;
      unsigned int    _startingBlock;
      int             _retries;            /* how many times to retry */
      IOStatus        _status;             /* filled by the concrete driver */
      Semaphore *    _notifyUponCompletion; /* wakeup the process if != 0 */

      addCommandAtTheEnd( Command * );
      addCommandAfterTheEnd( Command * );
      Command * lastCommand();
      Command * nextCommand();

private:
      DiskCommand * _nextCommand;        /* form a linked list of commands */
      DiskCommand * _lastCommand;
};

.....

/*****************************************************************************
 * A DiskCommand to transfer _numberOfBlocks to/from the physical memory
 * described by the PhysicalMemoryChain * _chain
 *****************************************************************************/
class PMCDiskCommand : public DiskCommand{
public:
      PMCDiskCommand( type );
      unsigned int          _numberOfBlocks; /* How many blocks to transfer */
      PhysicalMemoryChain * _chain;
};

inline  PMCDiskCommand::PMCDiskCommand( type ): DiskCommand( type ){};

/*****************************************************************************
 * A DiskCommand to transfer a number of disk blocks from/to contiguous
 * physical memory starting at address _physicalAddress
 *****************************************************************************/
class ContiguousMemoryDiskCommand : public DiskCommand{
public:
      ...
      unsigned int   _numberOfBlocks; /* How many blocks to transfer */
      char *         _physicalAddress;
};

....
```

Figure 6.1: The *DiskCommand* class and its subclasses

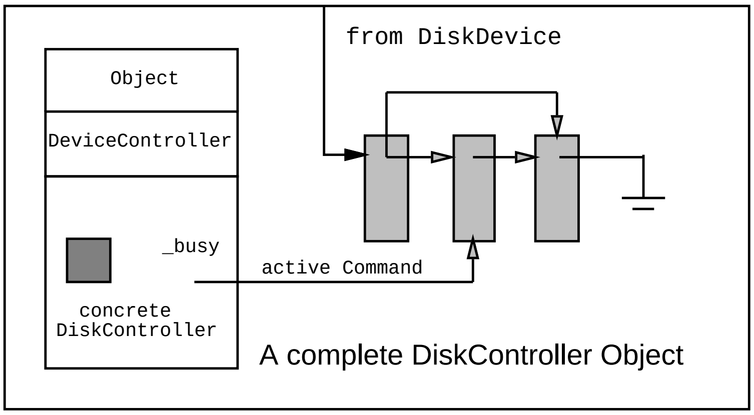


Figure 6.2: An I/O request waiting to be executed by the disk controller.

- `_status` Upon completion of the Input/Output operation dictated by this command the DiskController will fill this field. I discuss later what actions of the DiskDevice are dependent on the value of this field.

- Finally, the field `_notifyUponCompletion` is a placeholder for the process waiting for this Input/Output transfer. If the field is NULL then a new semaphore is created and the process is put to sleep on the semaphore. Otherwise it is assumed that the process is already waiting on the semaphore. The second case happens when the process has been waiting in the past in the queue of the DiskDevice.

The **`sendCommand()`** method causes the controller to start immediately the operation. Since the DiskDevices keep track of the load of the *DiskController* it is guaranteed that the command will not cause any problem[1]. The **`sendCommand()`** method blocks until the command is finished.

When the controller is busy doing Input/Output there is always one **active** *DiskCommand*. The active *DiskCommand* is the transfer being executed by the controller. A typical configuration is shown in Figure 6.2. Upon completion of the active command the interruptHandler of the class will call the method **`DiskDevice::commandCompleted()`** to notify the DiskDevice that the operation completed.

[1] We assume here that each Input/Output transfer is proceeded by a seek command. If the controller does not support simultaneous seek and transfer commands it should simply return immediately. The returning thread is responsible for starting an Input/Output transfer if the controller is not busy.

| *DiskDevice* | *Device* | |
|---|---|---|
| | DiskDevice | ( memTransferType, numOfUnits, log2UnitSize, retries ) |
| | ~DiskDevice | () |
| IOStatus | read | ( start, numOfBlocks, PhysicalMemoryChain ) |
| IOStatus | write | ( start, numOfBlocks, PhysicalMemoryChain ) |
| int | numberOfUnits | () |
| int | unitSize | () |
| ... | ... | ... |
| IOStatus | getCommand | ( Command * ) |
| void | commandCompleted | ( Command * ) |

Table 6.1: The protocol of the DiskDevice class.

Special care is taken in case of errors. When the DiskController code returns a command with an error condition the field `_retries` is checked and if it is greater than zero it is decreased by one and the operation is tried again. If it is zero the field `_status` of the command is set to indicate the problem.

## 6.2 The DiskDevice Class

The class DiskDevice acts as a server for users of the device. It collaborates with a disk controller class to accomplish its responsibilities. The DiskDevice has three responsibilities. The first is to know about the disk it represents. This responsibility is inherited by the *Device* superclass, which as I mentioned in section 5.2, is responsible for keeping data related to the device it represents.

The second responsibility is to translate the requests of the user into requests the controller can handle. The client of the disk driver runs as a *Choices Process* in a Domain[2]. The user allocates a contiguous region in its Domain, large enough to fit all the disk blocks it wants to read or write. Then he creates a PhysicalMemoryChain based on this region of the virtual memory. The PhysicalMemoryChain is a description of frames in the physical memory corresponding to a region in the virtual memory of a Domain. The construction of the PhysicalMemoryChain also takes care of locking these frames in the physical memory. Hence, the pager process will not attempt to page them out while Input/Output is taking place. A contiguous region in the

[2] Recall from section 3.1 that a *Choices* Domain is a virtual memory address space.

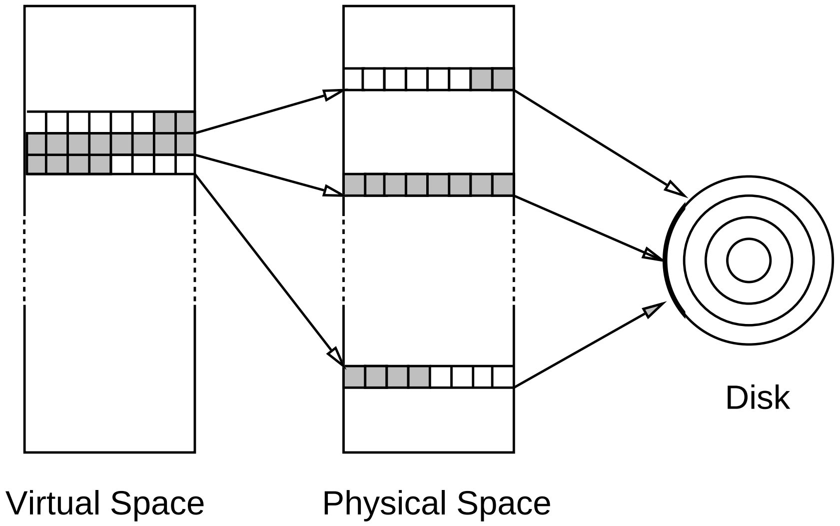


Figure 6.3: An example of a contiguous section of memory in a virtual memory address space mapped to a discontiguous section in the physical memory address space.

virtual memory space of the process might correspond to a discontiguous region in the physical memory address space. For instance, figure 6.3, shows a transfer of 14 disk blocks from a virtual address space to a disk. Each physical page frame is supposed to be 8 disk blocks large. After creating the PhysicalMemoryChain the process fills one PMCDiskCommand (see Figure 6.1) and sends the command to the DiskDevice.

Disk Controllers that support either scatter-gather DMA or DVMA(see section 4.4) can directly use the PhysicalMemoryChain. However, other controllers cannot handle transfers to and from discontiguous areas in the memory. For instance, the IBM Micro Channel bus[26] supports DMA transfers but does not support DVMA. In other cases the internal structure of the disk should be taken under consideration. For example, in some disks, a contiguous transfer that crosses a cylinder boundary, cannot be performed in one operation. The DiskDevice is responsible for splitting the original *DiskCommand* into a list of *DiskCommands* that can then be sent to the disk controller class. They are then handled as described in the previous section.

The third responsibility of the DiskDevice is to queue Input/Output requests for a device while the device is busy[3]. *Choices* is a multithreaded system. Hence, several threads of control

[3] I assume that each controller can perform a seek command for each device along with only one data transfer in parallel. This is the case with nearly every disk controller[48].

might concurrently send an Input/Output request for the same device. At any time a DiskDevice is in one of four states:

- idle without having any requests in the queue.
- waiting completion of a seek being performed by this device.
- idle having Input/Output request in the queue.
- waiting completion of an I/O request by this device.

The DiskDevice starts an Input/Output request by first sending a seek request to the disk controller object. Then, the controller is responsible for requesting further commands by calling the method `DiskDevice::getCommand()`. If the DiskDevice has any commands it returns the first command in the queue. Otherwise it returns a NULL value. The disk controller object is also responsible for informing the DiskDevice object when the operation is completed by calling the method `commandCompleted()`. There are two reasons why the DiskController continues the operation initiated by the DiskDevice. The first reason is that the Device is not in a position to know if the controller is busy. The second reason is that controllers for minis and mainframes can concurrently handle I/O transfers from more than one *Devices*. Since the DiskController continues the operation, the DiskDevice does not have to know the properties of the DiskController.

Although the DiskDevice is a concrete class it can be subclassed to get DiskDevices that keep more information about a particular disk. Also, it can be subclassed to provide more sophisticated queuing for Input/Output requests of the device. For instance one subclass of the disk might implement a Shortest-Seek-Time-First or a SCAN disk scheduling policy[40, 13].

Currently, two DiskDevices are supported. The one is the class DiskDevice(see Table 6.1) that knows nothing about the internal structure of the disk. The other class is the class DetailedDiskDevice(see Table 6.2) that knows how the disk is partitioned in cylinders, heads and sectors. Such information is necessary for performing disk scheduling and for splitting *DiskCommands* that cross cylinder boundaries. This section describes the protocol of the class DetailedDiskDevice. However, the responsibilities of this class are split between this class and its superclass. For example, the DiskDevice is responsible for splitting PMCDiskCommands to

| *DDDevice* | *DiskDevice* | |
|---|---|---|
| | `DDDevice` | `( memTransferType, diskTransferType, numOfCyls, tracksPerCyl,secsPerTrack, log2UnitSize, retries )` |
| | `DDDevice` | `()` |
| `IOStatus` | `read` | `( start, numOfBlocks, PhysicalMemoryChain )` |
| `IOStatus` | `write` | `( start, numOfBlocks, PhysicalMemoryChain )` |
| `int` | `diskFullAddress` | `( unit, & cylinderNum, & headNum, & sectorNum )` |

Table 6.2: The protocol of the class DetailedDiskDevice.

ContiguousMemoryDiskCommands while the DetailedDiskDevice is responsible for splitting the commands even further so as to overcome cylinder crossings if necessary.

### 6.2.1 Constructing a DetailedDiskDevice

The constructor of a DetailedDiskDevice takes a number of arguments that contain information about the device it represents. Such information consists of the number of disk units per track, the numbers of tracks per cylinder, the number of cylinders on the disk and the size of the unit of the device in bytes.

The most important arguments of the constructor of a DetailedDiskDevice are the arguments **`memoryTransferType`** and **`diskTransferType`**. The argument **`memoryTransferType`** takes two values, `ContiguousMemoryTransfer` and `DiscontiguousMemoryTransfer`. The value `ContiguousMemoryTransfer` means that each read or write operation should be split into one or more ContiguousMemoryDiskCommands. The argument **`diskTransferType`** also takes two values, **`crossCylinders`** and **`notCrossCylinders`**. The value **`crossCylinders`** means that a single command can cross cylinders. The value **`notCrossCylinders`** means that a single command cannot cross cylinders and should be split.

### 6.2.2 Translating a Disk Block Address

The **`fullDiskAddress()`** translates disk unit addresses to **`(cylinder, head, sector)`** triples. Not all the disk controllers need such a translation since most of the controllers perform this translation in hardware. However, the translation is necessary to find if a data transfer crosses cylinder boundaries or to implement disk scheduling policies. The method **`DiskDevice::fullDiskAddress()`** performs a default translation shown in Figure 6.4. Subclasses should overload this method to perform other translations.

```
IOStatus
DiskDevice::fullDiskAddress(int diskUnitNum, int & cylinderNum,
                                        int & headNum, int & sectorNum ){

      cylNum =      diskUnitNum / _unitsPerCylinder  ;
      headNum =   ( diskUnitNum % _unitsPerCylinder ) / _unitsPerTrack;
      sectorNum = ( diskUnitNum % _unitsPerTrack ) + 1;

}
```

Figure 6.4: The method `DiskDevice::fullDiskAddress()`.

### 6.2.3 Methods of a DetailedDiskDevice

The invocation of the `DiskDevice::read()` method causes the construction of a number of *DiskCommands*. In the trivial case that the requested operation can be handled by the DiskController one *DiskCommand* is built and scheduled. In other cases more than one *DiskCommands* can be built and scheduled. The same procedure applies to the `write()` method. Because of the common handling of the `read()` and `write()` method, each such operation is forwarded to a protected `doio()` method.

The methods `numberOfUnits()` and `unitSize()` return information about the structure of the disk. Since this information is part of the state of the DiskDevice no *Commands* are sent to the DiskController. The *DiskDevice* should also have other methods like `format()` that I do not discuss here.

## 6.3 A Detailed Example

The DiskController and *DiskDevices* work together in order to carry out I/O requests. An example sequence is graphically presented in Figure 6.5. In this example a disk controller controls two DiskDevices, the devices D0 and D1. The DiskController object is shown as a large rectangle while the DiskDevices are shown as small circles. A drawing representing a busy DiskDevice or DiskController is filled with a shadowed pattern. A drawing representing an idle DiskDevice or DiskController is filled with a white pattern. Small rectangles represent *DiskCommands*. Each request has a sequence number which reflects the order with which the requests entered the system. Drawings representing Input/Output requests for D0 are filled with a black pattern. A dotted line from the disk controller object to the DiskDevice means that the controller is performing a seek request on this device. A solid line from the disk

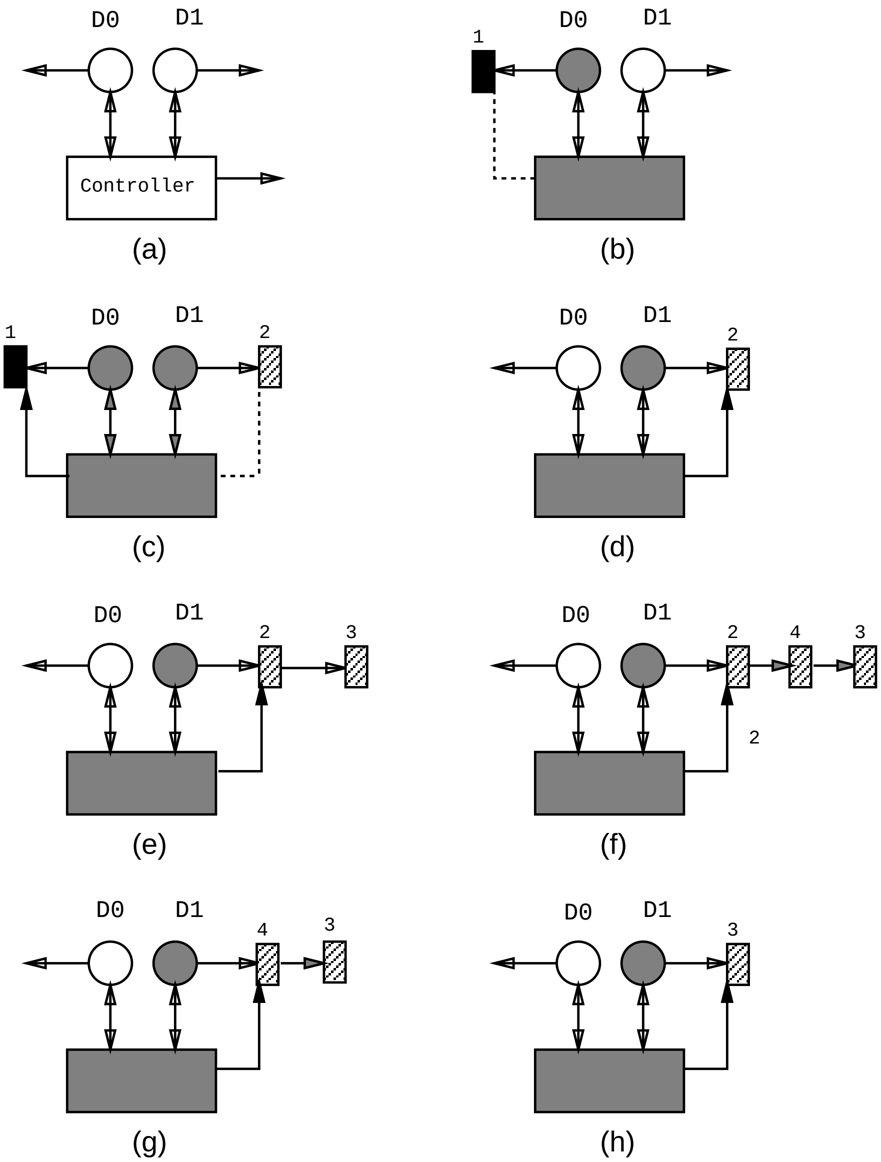


Figure 6.5: An example that shows how I/O requests are moved inside a system consisting of a DiskController object and two DiskDevices.

controller object to the *DiskDevice* means that an Input/Output transfer is taking place for this device.

Originally the system is empty as depicted in Figure 6.5.(a). The method `:read()` is invoked on D0 which causes the *DiskCommand* number 1 for D0 to be built. Since the device is not busy a SeekCommand is sent to the DiskController. The seek command is initiated and the `seekCommand()` blocks(see Figure 6.5.(b)). Eventually the seek command is completed and the interrupt handler calls the method `getCommand()` of the DiskDevice object to get and start executing the transfer request. While the I/O is taking place the `write()` method is invoked on D1 causing the construction of the *DiskCommand* number 2. Since the device is not busy, a SeekCommand is sent to the DiskController. The seek request is initiated and the `sendCommand` blocks (see Figure 6.5.(c)). Let's assume that the seek command completes before the command number 1 completes. The interrupt handler will inform the DiskDevice D1 that the command completed but will not call the `DiskDevice::getCommand()` since the controller is busy serving another request.

Eventually, the request number 1 is completed and the device interrupts by invoking the method `interruptHandle()` on the DiskController class. The DiskController calls the method `DiskDevice::CommandComplete()` which informs the DiskDevice D1 that the command completed. Since DiskDevice D1 has a pending transfer request, the DiskController calls the method `getCommand()` and starts executing the transfer(see Figure 6.5.(d)).

While the request number 2 is executing the *DiskCommand* number 3 is built in the DiskDevice D1 as a result of another method invocation. Since the device is busy the request is queued(see Figure 6.5.(e)). Next, the `read()` method is invoked on D1, causing the *DiskCommand* number 4 to be built. Since the device is still busy executing the request number 2 the request number 4 is also queued. Notice that for seek optimization reasons the command number 4 is entered before the command number 3(see Figure 6.5.(f)). Eventually, the command number 2 completes and the waiting thread resumes execution. The interrupt handler starts a seek operation for the command number 4(see Figure 6.5.(g)).

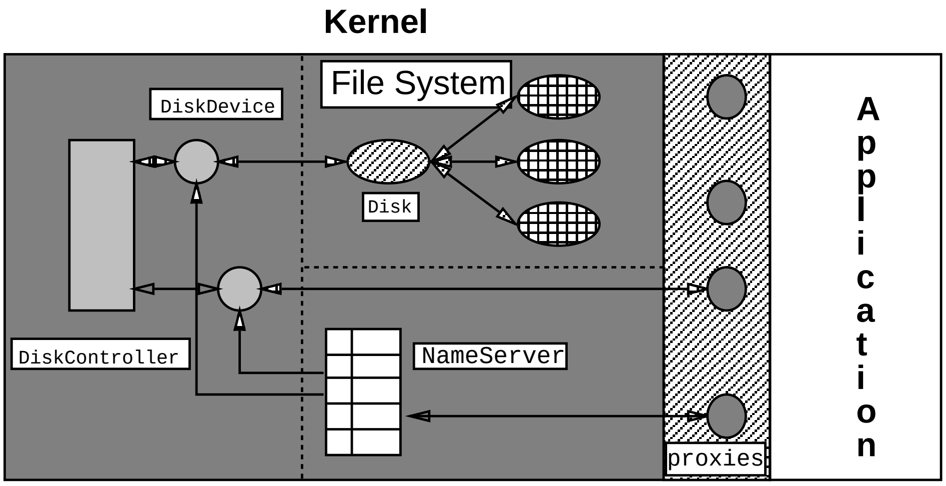


Figure 6.6: An example that shows how the disk subsystem integrates with other parts of the kernel system.

## 6.4 The Disk Driver and the *Choices* File System

The *Choices* file system expects disks to be subclasses of the class *PersistentStore*. DiskDevices are supported (see section 5.2.2) by the class Disk. The class Disk is a *PersistentStore* that reads and writes by sending commands to a DiskDevice.

Since the Disk supports DiskDevices a DiskDevice can be converted to a Disk as I discussed in section 5.2.2. When the driver is installed its devices are attached and bound to the name server. The user can then convert them to *MemoryObjects* and uses them through the file system. Figure 6.6 shows how the disk subsystem interacts with the file system and the kernel name service. In this example a controller has two disks. One of the DiskDevices has been converted to a *Disk*. The other disk is accessed through a proxy object by a user. This way the user can format or make a file system on the disk.

## 6.5 Summary

This chapter describes how the device management framework can be extended to support disk device drivers. A disk device driver consists of a machine dependent DiskController class and some DiskDevice classes. The DiskDevice class provides an interface to be inherited by *Devices* representing disks. Its default behavior is to accept requests from the user, transform the

```
FiSh> ns DiskDevice
>fd0          DetailedDiskDevice[0x270440]{2}
>rd9          ContiguousDiskDevice[0x263600]{2}
>hd0          BufferingDiskDevice[0x484000]{3}
>hd1          BufferingDiskDevice[0x508000]{3}
FiSh> inspect hd0
Device 0 of ESDIDiskController[0x27a080]
NumberOfUnits: 225280  UnitSize: 512
This is BIOSPartitionsContainer[0x58e800]{1}
Partition:  0 Type:    AIX bootable  Start Unit:      32 Length:    4103
Partition:  1 Type:      AIX normal  Start Unit:   72616 Length:  152664
Partition:  2 Type:         Unknown  Start Unit:       0 Length:       0
Partition:  3 Type:         Unknown  Start Unit:       0 Length:       0
The Device can be converted to objects of the following classes:
Class[0x1f1e68]{2}(Disk)
```

Figure 6.7: An example of a number of disk related commands from the *Choices* `FiSh` on the PS/2 implementation.

requests to commands and send them to the DiskController. The DetailedDiskDevice subclass is intended to be used with more primitive disk controllers which cannot handle multiblock transfers to discontiguous physical memory or transfers that cross cylinder boundaries.

I used this framework for the IBM PS/2 implementation of *Choices*. The PS2DisketteController is a *DevicesController* representing the hardware on the motherboard that communicates with up to two diskette drives. The PS/2 diskette controller is a simple controller that cannot handle transfers to discontiguous memory or transfers that cross cylinder boundaries. Therefore, it uses DetailedDiskDevices to represent diskette drives. The ESDIDiskController is a class representing a disk controller for ESDI disks. Such controllers are boards that are installed in the expansion slots of the PS/2 computer. Each PS/2 computer can have up to two ESDI disk controllers. Each ESDI disk controller can have up to two disks. ESDIDiskControllers are more sophisticated controllers than the PS/2 diskette controllers. However, they use the Micro Channel DMA which cannot handle transfers to discontiguous regions of physical memory. Hence, it reuses the DetailedDiskDevice.

Figure 6.7 shows an example DiskDevices bound to the name server. The BufferingDiskDevice is an experimental DiskDevice. I built this class to evaluate how buffering at the disk level affects the performance of the system. In this particular case the BufferingDiskDevice is used to represent an ESDI disk. It is a good example of how the separation of the *Devices* and the *DevicesControllers* leads to more flexible systems. The *DiskDevice* **`rd9`** represents the physical memory of the computer as a *DiskDevice*. It corresponds to the **`/dev/mem`** files of UNIX systems.

# Chapter 7

# Reusable Character Device Drivers

This chapter discusses an extension of the device drivers framework that supports character devices. A character device is a device that can transfer characters over a communication line. In contrast to network and disk devices, which transfer many bytes in each operation, character devices transfer only a few bytes in each operation. Serial line and keyboard ports are typical examples of character devices.

Monolithic drivers for character devices are usually very complicated because a character device driver has a large number of responsibilities. The design I propose in this chapter splits the responsibilities of a driver for character devices into three levels as shown in Figure 7.1. At the bottom level there are *DevicesController* objects whose responsibilities and protocol are described in the first section of this chapter. At the middle level there are objects of class CharacterDevice which I describe in the second section of this chapter. Finally at the top level there are objects of class *InputStream* and *OutputStream* whose responsibility I describe in the third section of this chapter. This seperation allows reuse of code among drivers of different character devices.

## 7.1 The Controllers of Character Devices

The objects that represent character devices controllers are instances of machine dependent classes, subclasses of the *DevicesController* class. Each system should have one *DevicesController* class for every different type of character controller that the system supports. A *DevicesController* class representing controllers of character devices has three responsibilities.

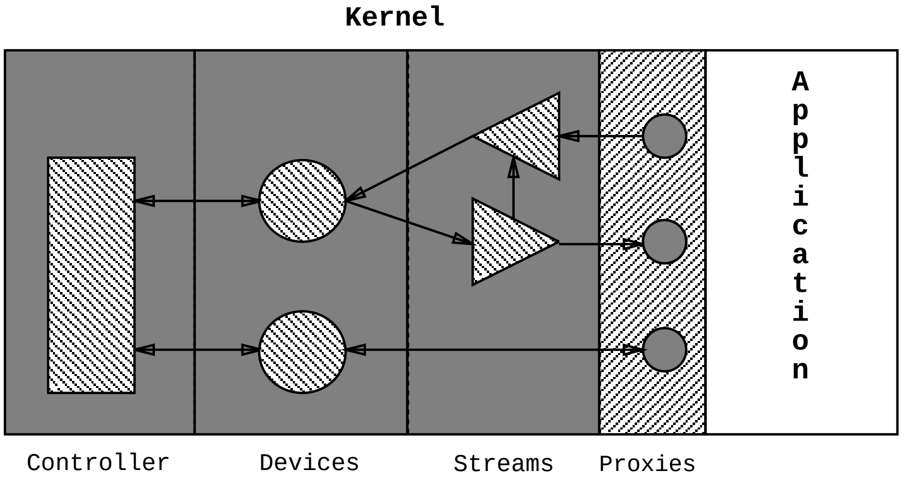


Figure 7.1: An overview of the Character Input/Output System.

1. The first responsibility is to overload the `DevicesController::attach()` method. As I discussed in section 5.1.3 this method returns properly initialized *Devices*. In this case, CharacterDevices are returned. For instance, a serial line controller returns a SerialLineDevice while a keyboard controller returns a KeyboardDevice.

2. The second responsibility is to know how to output or input strings of characters and notify the corresponding CharacterDevice.

3. The third responsibility is to implement certain control commands like the ones shown in Figure 7.2.(b) for a serial line controller.

The invocation of the **`sendCommand()`** method on a character *DevicesController* with an **`OutputStringCommand`**(see Figure 7.2) as an argument causes the beginning of the execution of the output command. The *DevicesController* does not check if the device is busy since this checking is done by the CharacterDevice which invoked the method. As shown in Figure 7.2 the **`OutputStringCommand`** takes two arguments. The one argument is the string of characters to transfer. The other argument is the length of the string.

Input to character devices comes asynchronously. For instance, a character arrives whenever a user presses a key on the keyboard. When a string of characters is received by the hardware the system is interrupted. As I discussed in section 5.1.6, the *DevicesController* installs a DevicesControllerException to handle this interrupt, when it is initialized. Upon arrival

```
class OutputStringCommand : public Command{
      char * string;
      int length;
      Command * next;
};
                  ( a )


class setBaudRate : public Command{
      int baudRate;
}

class setParityCommand : public Command{
      int parity;
}

class setCharLengthCommand : public Command{
      int charLength;
}
                (b)
```

Figure 7.2: A number of commands used in the interface between the CharacterDevice and its *DevicesController*.

of a string, the interrupt handler of the *DevicesController* calls the method **CharacterDevice::stringArrived()** and passes the string. The next section describes how the CharacterDevice handles the input.

This kind of partition of responsibilities between the *DevicesController* and the CharacterDevice allows the system programmer to concentrate on the machine dependent details of the controller. The machine independent responsibilities are carried out by the CharacterDevice class and the stream classes discussed in the next two sections.

## 7.2 The Class CharacterDevice

Instances of the class CharacterDevice represent devices associated with character controllers. The main characteristic of such devices is that they can transfer characters across communication lines. A character device is able to input or output strings of characters. Upon the completion of an output operation, the device asynchronously notifies the system by interrupting the normal flow of control. In the same way the device notifies the system upon arrival of a character or possibly a string of characters.

The class CharacterDevice has three responsibilities.

| *CharacterDevice* | *Device* | |
|---|---|---|
| | `CharacterDevice` | `( int numberOfChars, int flags )` |
| | `~CharacterDevice` | `()` |
| `IOStatus` | `outputString` | `( char *, int length, Priority )` |
| `IOStatus` | `setInputHandlingFunction` | `( (*f)(Obj*, char *), Obj* ob )` |
| `IOStatus` | `blockOutput` | `()` |
| `IOStatus` | `unblockOutput` | `()` |
| `IOStatus` | `flushOutput` | `()` |
| `IOStatus` | `getCommand` | `( Command * )` |
| `void` | `commandCompleted` | `( Command * )` |
| `void` | `stringArrived` | `( char *, int len )` |

Table 7.1: The protocol of the CharacterDevice class.

1. The first responsibility is to split strings of characters coming from clients of the CharacterDevice, and forward them to the controller.

2. The second responsibility of a CharacterDevice is to handle strings of characters that arrive asynchronously from the controller.

3. The third responsibility of a CharacterDevice is to queue output commands when the controller is busy processing an earlier request.

In order to carry out its responsibilities the CharacterDevice should know if the device is busy executing an operation or not. The next three subsections elaborate on these three responsibilities.

### 7.2.1 Sending Characters to a Device

The invocation of the method **`CharacterDevice::outputString()`** causes the beginning of an output operation. The method is usually invoked by an *OutputStream* class. If the device is busy the CharacterDevice queues the request and returns immediately. The operation will be started later by the interrupt handler of the controller. If the device is not busy the state of the CharacterDevice changes to busy and the output transfer is started by calling the method **`sendCommand()`** of the controller. Usually, character controllers cannot output more than some number of characters in one operation. For instance, serial line controllers usually accept only requests to output one character at a time. The CharacterDevice knows what is the maximum number of characters that the device can output at once. If a request contains a string with more characters, the CharacterDevice splits the operation in more than one output string commands. The format of such a command is shown in Figure 7.2.(a). If the `OutputString` command is sent to an input only controller, for instance a keyboard controller, the command will be rejected.

As shown in Table 7.1 each output operation has a priority associated with it. The priority parameter can take two values, `HIGH` and `NORMAL`. The CharacterDevice keeps two queues of outstanding requests. When the controller asks for the next command a command from the high priority queue is dequeued and given to the controller. If the high priority queue is empty then commands are dequeued from the normal queue. This feature allows a device to have more than one *OutputStreams* associated with it. One of these streams might be a high priority stream associated with system processes that should not be blocked by the user. For instance, if a user blocks the output to all the processes that share the system's console important messages might get lost.

Upon the completion of an output operation the machine dependent controller calls the method `CharacterDevice::commandCompleted()`. If the device has queued requests for output then the method `sendCommand()` of the controller will be called again. Otherwise the state of the device will change to idle. Whenever an `OutputStringCommand` is completed the buffer associated with it is deleted.

### 7.2.2 Receiving Characters from a Device

A controller notifies a CharacterDevice that a string of characters arrived by calling the method `CharacterDevice::stringArrived()`. The default behavior of the CharacterDevice is to discard the string. However, a client of the CharacterDevice can change this behavior by invoking the method `setInputHandlingFunction()` shown in Table 7.1. This method passes a function that is kept as part of the state of the CharacterDevice. Whenever a string arrives the device forwards the string to its client by calling this function. The receiver of the string is responsible for examining the input and taking the appropriate action.

The reason for not handling the arriving string inside the CharacterDevice is that most of the devices interrupt on every character arrival. Then, software should examine the character or characters arrived. Certain strings might have special meaning. For instance in UNIX, `CTRL-c` kills the current process and `CTRL-s` blocks the output. Under other circumstances echoing

| *CharDeviceInputStream* | *InputStream* | |
|---|---|---|
| | `CharDeviceInputStream` | `( CharDevice *, int flags )` |
| | `~CharDeviceInputStream` | `()` |
| `IOStatus` | `read` | `( char *, int len )` |
| `void` | `setOutputStream` | `( OutputStream * )` |
| `void` | `setMode` | `( int mode )` |
| `CharacterDevice *` | `device` | `()` |

Table 7.2: The protocol of the CharacterDeviceInputStream class.

should be enabled and disabled. Since such processing is not related to the controller or the device this responsibility is assigned to the client of the CharacterDevice.

### 7.2.3 Controlling the Output to a CharacterDevice

The methods **`blockOutput()`** and **`unblockOutput()`**, shown in Table 7.1, can be used to control the output to a CharacterDevice. When the first method is invoked the CharacterDevice stops outputting characters and waits for the **`unblockOutput()`** method to resume outputting. Output operations that arrive while the device is blocked are queued. These methods are usually invoked to the CharacterDevice in response to a **`Scroll Lock`** keystroke sent by the user. However, high priority output requests will still be sent to the controller. This way processes that open an *OutputStream* with a **`NODELAY`** flag will not be blocked by other processes that have opened another *OutputStream* on the same CharacterDevice.

The output of a character string occurs concurrently with the user's thread which initiated the operation. If the user wants to ensure that the output operation will be performed before proceeding he should invoke the method **`flushOutput()`**. The user who invoked this method blocks until the CharacterDevice has output all the requests that preceded the invocation of the **`flushOutput()`** method. If no waiting requests exists when the **`flushOutput()`** method is invoked then the method returns immediately.

## 7.3 The Character Device Stream Classes

The classes *InputStream* and *OutputStream* form the uniform Input/Output abstractions for *Choices* processes. Every application process has associated with it an *InputStream* and an *OutputStream*. A process might read characters from an *InputStream* that comes from a serial

| CharDeviceOutputStream | *OutputStream* | |
|---|---|---|
| | `CharDeviceOutputStream` | `( CharDevice *, int flags )` |
| | `~CharDeviceOutputStream` | `()` |
| `IOStatus` | `write` | `( char *, int len )` |
| `IOStatus` | `flush` | `()` |
| `IOStatus` | `blockOutput` | `()` |
| `IOStatus` | `unblockOutput` | `()` |

Table 7.3: The protocol of the CharacterDeviceOutputStream class.

line, from a file in a disk, or from a string of characters in memory. Subclasses of the *Stream* classes are responsible for knowing where and how they will perform the **`read()`** and **`write()`** operations.

The classes CharacterDeviceInputStream and CharacterDeviceOutputStream, briefly discussed in this section, are examples of classes that can stand between a CharacterDevice and the user. Other more complex mechanisms, like STREAMS[2, 27], could also be implemented on top of the CharacterDevice.

### 7.3.1 The CharacterDeviceInputStream Class

The responsibility of the CharacterDeviceInputStream class(see Figure 7.2) is to perform line buffering, echoing and character processing of the arriving characters. Usually a CharacterDeviceInputStream will be associated with a CharacterDeviceOutputStream. When a character arrives from the device the character will be examined. Normal characters will be echoed to the CharacterDeviceOutputStream and then buffered until a process asks for them. Special keystrokes might initiate operations like disabling the output and killing the current process. According to the settings of the *InputStream*, echoing may be disabled, different translations may be set and a user process might be notified for every arriving character instead of line buffering the characters.

### 7.3.2 The CharacterDeviceOutputStream Class

The class *OutputStream* is an abstract class. It provides the user with an outlet for character strings. When a process starts execution it looks in its domain name server for an *OutputStream* bound under the name `StandardOutput`. Class CharacterDeviceOutputStream is a subclass of *OutputStream*, which acts as a client of a CharacterDevice.

```
FiSh> inspect tty0
Device 0 of PS2SerialLineController[0x2704c0]
Baud rate:  9600 Bits Per Character:   8
Parity   :  none Stop Bits         :   1
The Device can be converted to objects of the following classes:
Class[0x1f1ed0]{2}(CharacterDeviceOutputStream)
Class[0x1f1f40]{2}(CharacterDeviceInputStream)
```

Figure 7.3: An example of inspection of a CharacterDevice object representing a serial line.

The main responsibility of the class CharacterDeviceOutputStream is to provide flow of control between the user and the device. When a user `writes()` a string, a new buffer is allocated, the string is copied to the buffer and sent to the CharacterDevice. If the user wants to wait until the output has been printed by the device he should sent the message `flush()`.

Flow of control is taken care of "smart" buffers. The invocation of the method `write()` causes the allocation of a new SmartBuffer. The string is then copied into this buffer. The allocation is done by invoking the overloaded operator `SmartBuffer::new( )`[47]. This operator cooperates with a buffer pool manager object to get a new buffer. If at the time a `write()` message arrives no available buffer exists, the process is blocked until output buffers become available. Output buffers become available when the controller has printed its output and `deletes` the buffer. The overloaded destructor of the buffer will return the buffer to the pool and will signal any processes that are blocked waiting for buffers. If the process that wants to output is a high priority process that should not be blocked the overloaded operator `new` will return immediately with a value of 0. The process that invoked the `write()` method will then be informed that the output operation failed.

## 7.4 Summary

This chapter gives an example of how the proposed framework for device drivers can be used to implement reusable drivers for character devices. A device driver for a character device has two parts. One part is machine dependent and represents the controller of the device. The other part is machine independent and represents the device itself. Difficult programming tasks like blocking and unblocking the output, enabling and disabling the echoing and line buffering can be handled in a machine independent way. So a significant part of the driver can be reused.

In addition, the system programmer who is developing the machine dependent controller driver can now concentrate on the machine and controller dependent details.

The most important advantage of this design is that a class at any level can be substituted by a new class. For instance, if the CharacterDeviceOutputStream does not provide all the required features a new class can be built. The new class can be subclassed by either the *OutputStream* or the CharacterDeviceOutputStream. The new class can then be dynamically linked to the system. The conversion mechanism which is based on double dispatching, as I explained in section 5.2.2, does not require any modification of the CharacterDevice class. The CharacterDevice can now easily be converted to an object of the new class.

A prototype implementation of this design was used when porting *Choices* on the PS/2 porting of Choices. A serial line controller with three attached serial lines, a keyboard and mouse controller shared the CharacterDevice and the stream classes. Figure 7.3 shows how the user can find the settings of a serial line which is bound to the name server under the name `tty0` from the Choices shell. Apart from the settings, the inspection shows the classes which support the *Device*(see 5.2.2).

# Chapter 8

# Summary and Conclusions

In this thesis I described a device management framework for an object-oriented operating system. Each device driver is structured as a module consisting of a *DevicesController* class and a number of *Devices*. An instance of a *Device* is constructed and bound to the NameServer when a device is added to the system. Each *Device* acts as a server for components of other *Choices* frameworks. In turn, most of the *Device*s act as clients of *DevicesController* objects. The protocol of a *Device* depends on the device it represents.

Instances of *DevicesControllers* classes represent hardware controllers. A *DevicesController* acts as a server for possibly several *Devices*. A *DevicesController* is not visible to the user of a device. I/O operations should only be requested from a *Device*. The only other framework that interacts with a *DevicesController* is the exception handling framework. The interface between a *Device* and a *DevicesController* is a message interface based on *Command* objects. User requests to a *Device* cause the construction of one or more *Commands* which are then sent to a *DevicesController* object using the `::sendCommand()` method. A message interface between the *DevicesController* and the *Device* has two advantages. The first is that a *Device* can be reused with different *DevicesControllers*. The second advantage is that the message interface does not force a *DevicesController* to have a specific interface that depends on its devices. The protocol of a *DevicesController* subclass can change without requiring a change to existing *Devices*. On the other hand, the message interface cannot be checked at compile-time to ensure consistency. I think that this is not a major problem, since the interface is internal to the framework.

Structuring a device driver as a module consisting of a *DevicesController* and a number of *Devices* has two advantages. The first advantage is that *Devices* can be reused with different machine dependent *DevicesControllers.* The second advantage is that the implementor of a device driver can now concentrate on the machine dependent details of the driver he implements.

The DevicesManager class is the third component of the framework. Each system has only one object of this class. When a *DevicesController* is loaded into the system it registers with the DevicesManager object. Hardware controllers and devices that are added to the system are also registered with the DevicesManager. The DevicesManager is informed of the addition and removal of hardware components by the system administrator or by the cooperation of hardware and machine-dependent software. The DevicesManager matches physical controllers with registered *DevicesControllers.* For each physical controller a "matching" *DevicesController* is instantiated. In addition, for each physical device a *Device* is constructed and returned when the method `DevicesController::attachDevice()` is invoked. The new *Device* is then bound to the NameServer.

Other frameworks use the Device Management framework with the help of classes that provide communication between the two frameworks. *Devices* are converted to objects in other hierarchies using the *Choices* conversion mechanism. Conversion is a term introduced in Smalltalk[20] and used for the collection classes. We generalize the conversion mechanism to apply to any class. We also combined the conversion mechanism with double dispatching[22] so that new inter-framework classes can be added to the system without changing existing classes inside the frameworks.

A device driver is probably the most machine dependent component of an operating system. However, I believe that there are a number of algorithms and techniques shared by many drivers in the same or different machines. My device management framework, presented in this thesis, implements this observation. I expect that the framework will be extended to include complete disk, character, display and network device management subframeworks similar to the models I presented in chapter 6 and 7.

## 8.1 Further Issues

There are three issues that I did not address in this thesis. The first issue is error logging. Devices are expected to fail due to hardware problems. In such cases, the *Devices* and *DevicesControllers* classes return an error status. However, more information must be given for the problem to be identified. In most operating systems this information is usually printed on the console. I think that an instance of an error logging class, possibly a subclass of the *OutputStream* is more suitable for an object-oriented system. The system administrator can then define the physical sink of the *OutputStream*.

The second issue I did not address is a general statistics mechanism. The system administator should be in a position to get statistics from several devices and controllers so as to figure out where the bottlenecks are and enhance the performance of the system. I think that a general uniform statistics mechanism for the device management framework should be available.

Finally, the third issue is the hardware protection of the address space of a device driver. It is a common practice to put all the parts of an operating system, including the device drivers, in one address space, leading to a monolithic kernel. Since operating systems become more and more complicated this approach has a number of disadvantages[4]. Recent research operating systems[50, 1, 8] have broken the traditional kernel into a microkernel and a number of modules that exist in different address spaces. The location of device drivers in a microkernel-based operating system is an open issue.

# Bibliography


[1] M. Acceta, R. Baron, W. Bolosky, D. Golub, R. Rashid, A. Tevanian, and M. Young. A new Kernel Foundation for Unix Development. In *Proceedings of the Summer 1986 USENIX Conference*, Atlanta, Georgia, July 1986.

[2] AT&T. *Streams Programmer's Guide.* Prentice Hall, Englewood Cliffs, New Jersey, 1989.

[3] Maurice J. Bach. *The Design of the UNIX Operating System.* Prentice Hall, 1986.

[4] Brian N. Bershad. *High Performance Cross-Address Space Communication.* PhD thesis, University of Washington, June 1990.

[5] Grady Booch. *Object-Oriented Design with Applications.* The Benjamin/Cummings Publishing Company, Inc, 1991.

[6] Paul L. Borril. High-Speed 32-bit Buses for forward-looking Computers. *IEEE Spectrum*, 26(7):34–37, July 1989.

[7] L. Cardelli and P. Wegner. On Understanding Types, Data Abstraction, and Polymorphism. *ACM Computing Surveys*, 17(4):471–522, December 1985.

[8] David Cheriton. The V Distributed System. *Communications of the ACM*, 31(3):314–333, March 1988.

[9] David R. Cheriton. UIO: a Uniform I/O System Interface for Distributed Systems. *ACM Transactions on Computer Systems*, 5(1):12–46, February 1987.

[10] Peter Coad and Edward Yourdon. *Object-Oriented Analysis.* Yourdon Press Computing Series, 1990.

[11] NCR Corp. *Understanding the Small Computer System Interface Systems.* Prentice Hall, Englewood Cliffs, New jersey 07632, 1990.

[12] O. J. Dahl and K. Nygaard. Simula, an algol-based simulation language. *Communications of the ACM*, pages 671–678, 1966.

[13] Harvey M. Deitel. *An Introduction to Operating Systems.* Addison-Wesley, Reading, Massachusetts, 1984.

[14] David W. Dykstra. Hardware Enforced Protection for Object-Oriented Operating Systems. Technical Report UIUCDCS-R-91-1666, University of Illinois at Urbana-Champaign, December 1990.

[15] David W. Dykstra. Object-Oriented Hierarchies Across Protection Boundaries. Technical Report UIUCDCS-R-91-1667, University of Illinois at Urbana-Champaign, February 1991.

[16] Janet I. Egan and Thomas J. Teixeira. *Writing a Unix Device Driver.* John Wiley And Sons Inc, 1988.

[17] Margaret A. Ellis and Bjarne Stroustrup. *The Annotated C++ Reference Manual.* Addison Wesley, 1990.

[18] Richard Rashid et al. Machine-independent virtual memory management for paged uniprocessor and multiprocessor architectures. In *Proceedings of the First International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 31–39, 1987.

[19] E. H. Frank and W. Mitch Bradley. *SBus Specification A.1.* Sun Microsystems Inc, 1990.

[20] Adele Goldberg and David Robson. *Smalltalk-80: The Language.* Addison-Wesley, Reading, Massachusetts, 1989.

[21] Jr. Harry Katzan. *Computer Organization and the System/370.* Van Nostrand Reinhold Company, 1971.

[22] Kurt J. Hebel and Ralph E. Johnson. Arithmetic and double dispatching in Smalltalk-80. *Journal of Object Oriented Programming*, March/April 1990.

[23] Bjorn A. Helgaas. Porting an Object-Oriented Operating System to the Macintosh II Family. Master's thesis, University of Illinois at Urbana-Champaign, May 1991.

[24] Richard Helm, Ian M. Holland, and Dipayan Gangopadhyay. Contracts: specifying behavioral compositions in object-oriented systems. In *Proceedings of OOPSLA '90*, pages 169–180, Ottawa, Canada, October 1990. Published and as SIGPLAN Notices, 25(10).

[25] John L Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach.* Morgan Kaufmann Publishers Inc, 1990.

[26] IBM. *PS/2 Hardware Interface Technical Reference.* IBM, 1988.

[27] Sun Microsystems Inc. *Streams Programming.* Sun Microsystems Inc, 1990. Part Number: 800-3826-10.

[28] Sun Microsystems Inc. *Writing device drivers.* Sun Microsystems Inc, 1990. Part Number: 800-3851-10.

[29] Intel. *80386 Programmer's Reference Manual.* Intel, 1986.

[30] John A. Interrante and Mark A. Linton. Run-time Access to Type Information in C++. In *Proceedings of the USENIX C++ Conference*, pages 233–240, San Francisco, California, April 1990.

[31] Ralph E. Johnson and Brian Foote. Designing Reusable Classes. *Journal of Object-Oriented Programming*, pages 22–35, June 1988.

[32] Ralph E. Johnson and Vincent F. Russo. Reusing object-oriented designs. Technical Report UIUCDCS-R-91-1696, University of Illinois at Urbana-Champaign, May 1991.

[33] Gerry Kane. *MIPS R2000 RISC Architecture*. Prentice Hall, Englewoods Cliffs, New Jersey, 1987.

[34] Samuel J. Leffler, Marshall Kirk McKusick, Michael J. Karels, and John S. Quarterman. *The Design and Implementation of the 4.3 BSD UNIX Operating System*. Addison Wesley, 1989.

[35] Douglas Eric Leyens. A Choices Implementation of the Universal Scheduling System. Master's thesis, University of Illinois at Urbana-Champaign, May 1989.

[36] Peter Madany, Roy Campbell, and Panos Kougiouris. Experiences Building an Object-Oriented System in C++. In *Procedings of Tools for Object Oriented Languages and Systems TOOLS'91*, France, March 1991.

[37] Peter W. Madany. An Object-Oriented Approach towards a General Model of File Systems: A Preliminary Exam Statement. Technical Report UIUCDCS-R-90-1607, University of Illinois at Urbana-Champaign, December 1990.

[38] Motorola. *MC68030, Enhanced 32-bit Microprocessor User's Manual*. Prentice Hall, 3rd edition, 1990.

[39] William F. Opdyke and Ralph E. Johnson. Refactoring: An aid in designing application frameworks and evolving object-oriented systems. In James TenEyck, editor, *Proceedings of symposium on Object-Oriented Programming Emphasizing Practical Applications(SOOPPA)*, pages 145–160, September 1990.

[40] James L. Peterson and Abraham Silberschatz. *Operating System Concepts*. Addison-Wesley, Reading, Massachusetts, 1985.

[41] Vincent Russo and Roy H. Campbell. Virtual Memory and Backing Storage Management in Multiprocessor Operating Systems using Class Hierarchical Design. In *Proceedings of OOPSLA '89*, pages 267–278, New Orleans, Louisiana, September 1989.

[42] Vincent Russo, Gary Johnston, and Roy H. Campbell. Process Management and Exception Handling in Multiprocessor Operating Systems using Object-Oriented Design Techniques. In *Proceedings of OOPSLA '88*, pages 248–258, September 1988.

[43] Vincent F. Russo. *An Object-Oriented Operating System*. PhD thesis, University of Illinois at Urbana-Champaign, October 1990.

[44] Cypress Semiconductor. *SPARC's User Guide*. Cypress Semiconductor, 2nd edition, 1990.

[45] Marc Shapiro. Structure and Encapulation in Distributed Systems: The Proxy Principle. In *Proceedings of the 6th International Conference on Distributed Systems*, May 1986.

[46] Harold S. Stone. *High-Performance Computer Architecture*. Addison-Wesley, October 1987.

[47] Bjarne Stroustrup. The Evolution of C++: 1985-1989. In *Proceedings of the USENIX C++ Conference*, Santa Fe, New mexico, November 1987.

[48] Andrew S. Tanenbaum. *Operating Systems, Design and Implementation.* Prentice-Hall, 1987.

[49] Andrew S. Tanenbaum. *Structured Computer Organization.* Prentice-Hall, 3rd edition, 1991.

[50] Andrew S. Tanenbaum, Robbert Van Renesse, Hans van Staveren, Gregory J. Sharp, Sape J. Mullender, Jack Jansen, and Guido Van Rossum. Experiences with the Amoeba Distributed Operating System. *Communications of the ACM*, 33(12):46–61, December 1990.

[51] Peter Wegner. Dimensions of Object-Based Language Design. In *Proceedings of OOPSLA '87*, pages 168–183, Orlando, Florida, October 1987.

[52] Rebecca Wirfs-Brock, Brian Wilkerson, and Lauren Wiener. *Designing Object-Oriented Software.* Prentice Hall, 1990.