1 Introduction

In this book, *a* ***small, communicating computer*** *(**SCC**) is a part of any electronic device that is implemented using the idea of a miniature stored program computer in conjunction with a communications device* (stored program computers are reviewed in Section 1.2). Figure 1-1 illustrates several classes of SCCs that have evolved during the last decade or so. In this characterization, you can see that mobile computers and smart cell phones are SCCs, but small computers that aren't mobile at all (like a TV set-top box) are also SCCs. The key characteristics of the computers in this class are that they are small enough that they *could* be mobile, and that they are capable of exchanging information with other computers.

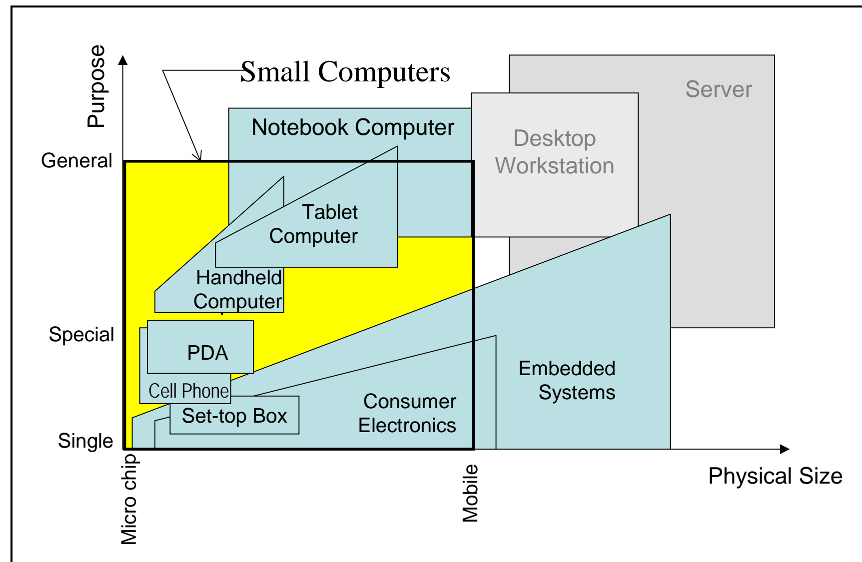


Figure 1-1: Small, Communicating Computers

The meaning of the axes in the figure are intended to be intuitive rather than precise. It also intentional that the boundaries of the SCC space are a little fuzzy. The "Purpose" axis differentiates among computer systems that can be used for a wide variety of applications, depending on how they are programmed, versus those systems that are designed to solve only a particular problem. The "Physical Size" distinguishes small computers from large ones: The arbitrary size marking of "Mobile" doesn't mean that SCCs must be mobile, it just means that they are small enough to be mobile.

Modern *notebook* (also called *laptop*) computers are at one extreme (in size and purpose) of SCCs: Today these machines are almost as powerful as desktop machines. Because some notebook computers are not really different from some desktop computers, we arbitrarily categorize notebook computers as not being wholly contained in the SCC space. Even so, every notebook computer differs from conventional desktop and server machines as follows:

- They are easily transported from one location to another.
- Their operating speeds tend to be a lower than desktop computers.
- They are configured with more specialized hardware devices (such as PCMCIA devices and wireless networking).
- They can operate from a battery, and have power management software that dims the screen and/or powers down the hard disk after a period of inactivity.

Nevertheless, notebook computers often use the same OS as a desktop computer. That is, operating systems and other system software can be configured so that it is well-suited for use with a large notebook or a desktop computer, for example, by including appropriate device drivers and enabling power management for the notebook environment.

Creative Commons Attribution-Share Alike 3.0 U.S. License

Large ("server") machines may also use the same OS as the notebook and desktop, although their *configurations* are distinct from the desktop versions. For example, Microsoft uses the same code for workstation and server versions of Windows XP and Vista but configures things like the memory management strategy differently in the two classes of machines. Servers can be general purpose timesharing computers, or specialized service providers: For example, a "cycle server" is an informal name for a network server computer whose primary purpose is to execute programs on behalf of other computers on the network. A cycle server is often used to compile multi file programs or to solve a system of differential equations. By contrast, a database server is specialized to interpret queries and then to retrieve information satisfying the query.

SCCs operate in a network environment, meaning that each SCC depends on (or provides services to) other computers via the interconnecting communication mechanism. During their interaction with other computers, SCCs employ the *client-server* communication model (explained in Chapter 3) usually by behaving as a client that requests services from network server machine – sometimes from general purpose servers that have more resources than the SCC, and sometimes from specialized servers for computing cycles or storage. As an example an SCC might playback a movie that is stored on a specialized storage server by only retrieving small portions of the movie at a time, thus avoiding the need to have enough storage to hold the entire movie at once.

Considering some of the specific types of computers in Figure 1-1, *tablet computers* are a variant of notebook computers – they differ from conventional notebooks in that they allow the user to enter information into the machine using a stylus and touch-sensitive screen. In terms of their input devices, they are a cross between a conventional laptop computer and a PDA (such as a Palm Treo or Compaq iPaq).

Handheld computers such as the Sony VAIO UX computers, Nokia Internet Tablet, the (now obsolete) Toshiba Libretto, and the Vulcan mini-PC (FlipStart) represent another interesting class of SCCs, albeit one that is not very visible in the current marketplace. They are remarkably similar to notebook computers, but on the practical side of things, they are distinctly different: The variety of devices are more limited than for notebooks, and they tend to have slower processors and less memory. Like tablet computers, handhelds can also have new kinds of devices that distinguish them from notebooks, such as audio input and output. There are not very many contemporary examples of handheld computers, but there is high potential for a new wave of these to appear at any time (depending on how many of them manufacturers think they can sell).

PDAs, *cell phones*, and *consumer electronic devices* (such as web appliance like the now-obsolete WebTV, or an email appliance like the Blackberry device) are designed to provide *specialized services*, so they are distinguished from the notebook/table/handheld family. These devices generally do not support a software development environment, and have more limited OS software and hardware than their general purpose relatives.

Wireless sensor network (*WSN*) *nodes* are specialized computers that are envisioned as being as small as dust particles at some time in the future (some researchers call them "smart dust"). WSN nodes have very small batteries, a few devices for sensing information about their environment (such as temperature, humidity, or vibration frequency) and a radio-based wireless network. A WSN node exists in a network with conventional computers, so WSN applications are typically designed from scratch as distributed applications.

Set-top boxes have been used by the cable and satellite TV industry for many years. The original purpose of a set-top box was to be an electronics device that converted a cable signal into one that could be played back on a (then) conventional television set. Television manufacturers quickly adopted this technology and began building "cable ready" TV sets. The second purpose of a set-top box is to provide a mechanism to decrypt an encrypted broadcast signal. This is useful in the case that the cable company wants to charge a premium for certain programming. Today, set-top boxes are designed to provide various additional services such as a companion web browser for the program that is currently being played. Forward thinkers in this industry even imagine set-top boxes providing special effects, programming guides, and so on.[1]

Generally you can think of an *embedded system* as an SCC that is a component in some larger system. The computer may be completely transparent to the larger system user: For example, an embedded system might control a programmable thermostat in a home, a sprinkler system, a microwave oven, a hard disk, or

[1] In 1995 [Fuhrt, et al, 1995] wrote about future features of set-top boxes. More than a decade later, some have come to pass, and others remain a dream.

Creative Commons Attribution-Share Alike 3.0 U.S. License

a rocket guidance system. Because embedded systems can become quite large and sophisticated, our informal characterization in the figure shows that some of them are just too large to be thought of as SCCs.

The remainder of this chapter provides background information you will find helpful for studying SCCs. If you are already comfortable with this information, you can skip right to Chapter 2 to begin learning more about distributed systems.

1.1 Operating Systems and Abstraction

> An **OS** is the part of the software that interacts directly with the computer hardware to export an abstraction of the hardware behavior using an **application programming interface** (**API**).

Besides the OS, software can be loosely categorized as *other system software* (like network abstraction software or software to convert cursive writing into characters), and *application software*. Both of these kinds of software *use* the features implemented by the OS by invoking functions implemented in the OS. Sometimes we say that the OS *supports* these other types of software since that software invokes OS functions, but the OS does not call functions implemented by the application or other system software.

The operating system's **application programming interface**, or **API**, describes the function names and parameters for the set of functions provided by the OS. The collective functions on the API are carefully designed so that together they define an *abstract* or *virtual machine* (a logical software environment) that behaves more or less like a physical computer. For example, when a C/Unix program invokes the function `write(int fileID, char *buffer, int bufLen)`, the OS will output `bufLen` characters stored in the memory location specified by `buffer`, into a file specified by `fileID`. That is, the underlying virtual machine represents storage devices as files, and the `write()` is the virtual machine's instruction that causes data to be written into a specified file.

A simple OS is designed to create simple (abstract) interfaces to the computer's hardware components, and a complex OS may define a correspondingly more complex virtual machine. In either case, the OS is implemented on top of the hardware and exports functions by which application programs can control the hardware (see Figure 1-2) using the OS API (which is traditionally called the **system call interface**).

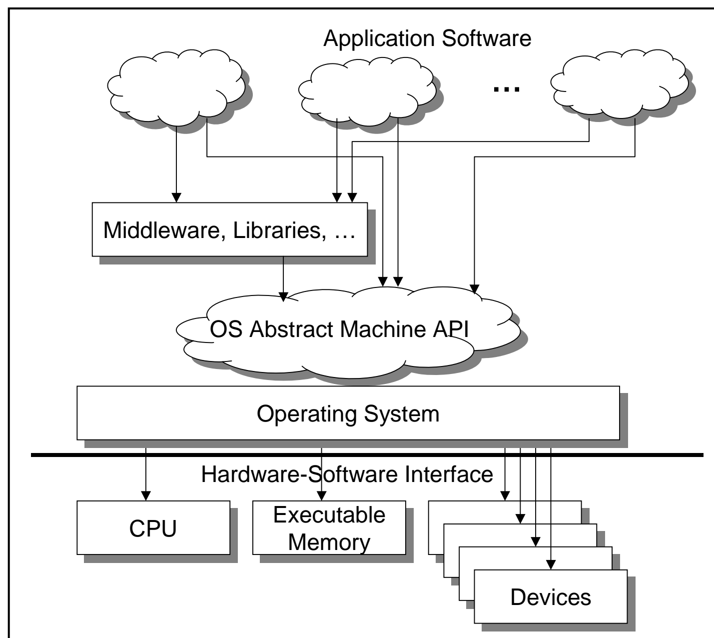


Figure 1-2: The OS Abstract Machine

Creative Commons Attribution-Share Alike 3.0 U.S. License

Let's take a closer look at abstraction. Here is a more descriptive example of how abstraction works: Today, many of the more general-purpose hardware display devices are based on *bitmaps* – a technology where a digital memory represents the image that will appear on the device. The physical screen is divided into rows and columns of small dots (called *pixels*) such that a unit of the memory is associated with each pixel. In order to create an image, the programmer writes information into specific locations of the memory to cause the corresponding pixels to display a color. To draw a red point in the center of a screen, the memory unit associated with the pixel in the center of the screen is written with a code corresponding to the color red. The physical terminal will then read this display memory and set the appropriate pixel to the color red.

Consider how an SCC displays the string "Hello, world" in the center of a bitmapped screen: If you programmed the display device without using any abstraction, you would have to determine the addresses of the pixels for the center of the screen, then how to represent the characters by setting the right pixels. The pseudo machine language instructions shown in Figure 1-3 represent the nature of the software that interacts directly with the display device to control its behavior (by storing data patterns into the display device bitmap). The code sets the color of each individual pixel using an instruction like

```
store     0xdf0002000, =8
```

which stores the number "8" into memory location `0xdf0002000`, thereby setting the display properties for the single pixel associated with that address. The figure also shows a code segment that uses the OS code for setting pixels; in this case, the programmer calls the OS `write()` function to copy a block of memory with the correct pixel settings directly to the display device. (In this simple example, our sample code does the hard work of character layout with the `convertString()` and `setCursor()` functions – which, realistically, would be library functions.) Finally, the C language `printf()` library function call illustrates an even higher level of abstraction, allowing the programmer to print the message on the display by specifying a minimum of information. The `cout` member function from the C++ `stream` class and the Java `System.out.println()` function from the `OutputStream` class are even higher abstractions than `printf()`.

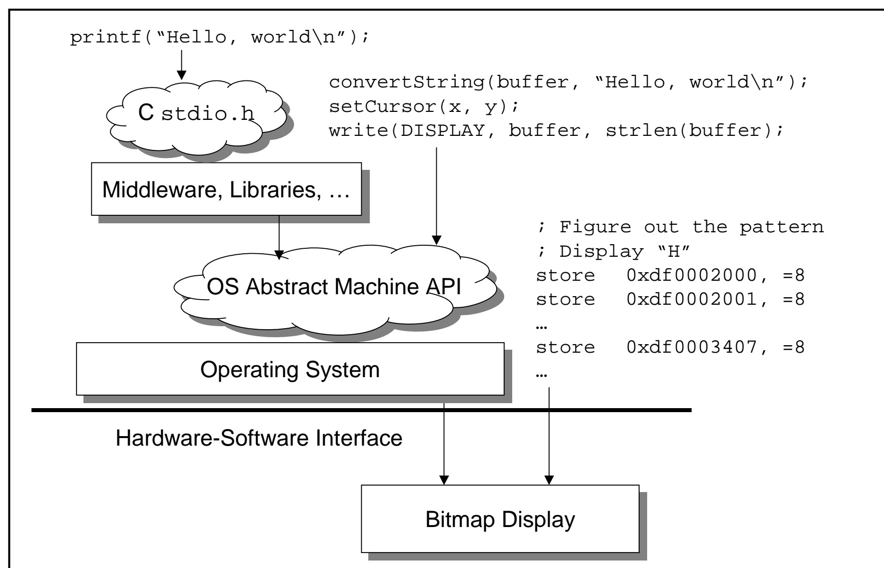


Figure 1-3: Writing to a Bitmap Display

Each OS exports its own API to define its own abstract machine; as suggested by the figure, generally the system software augments the OS API with library code (such as the C runtime library) to extend the OS abstraction. In the Windows family of operating systems, Microsoft used the Win32 API (about 2,000 functions) to define its virtual machine. Windows NT/2000/XP implement the entire API, Window 98/Me implements a subset of the API, and the early versions of Windows CE (now called Pocket PC) implemented about 500 of the Win32 API. Today, Microsoft is moving toward the .NET abstract machine to replaces (or at least provide an alternative) to the Win32 API.

Creative Commons Attribution-Share Alike 3.0 U.S. License

In the UNIX family, the IEEE **POSIX.1 API** specification[2] defines a standardized virtual machine. This allows Linux, OpenBSD, FreeBSD, Mac OS X, Mach, and others to all support the same POSIX virtual machine (with their own implementations). Both Windows and UNIX operating systems also support various C library functions, C++ libraries, and the Java Virtual Machine (JVM) meaning that programmers can write C, C++, and Java programs that use the respective abstractions in conjunction with the OS abstraction.

The complexity of the abstract machine varies from those that are only concerned with creating abstractions of hardware devices, to those that implement an elaborate virtual machine with a broad spectrum of abstract devices. This complex class of operating systems also handles a much more difficult problem: The virtual machine abstraction often requires that the OS create a technique for *sharing* the underlying abstractions that are parts of the virtual machine. For example, the operating systems for more general SCCs support multiprogramming – a technique by which a collection of executing programs share the use of the computer's processor and memory.

This series of booklets describes the spectrum of operating systems and technologies that are used in contemporary SCCs. This is an exciting area because there is a broad spectrum of technologies used in various systems, because the computers themselves use leading edge hardware technology, and because these computers exist as part of a *distributed computing environment*. That is, the software on these computers interacts with software that is executing on other computers in order to fulfill its application functionality.

By studying OS approaches for single-threaded dedicated systems, trusted process systems, then managed process systems, you will learn how the OS for any tiny embedded system is similar, and how it is different, from the OS used in a contemporary notebook computer.

1.2 SCC Hardware Basics

Binary stored computers began to be a reality in the 1940s, although the essential idea for stored program computers was actually described by Charles Babbage in 1837.[3] By 1940, various research groups had begun to build electronic computers, Randell's books describes all of these early efforts.

For the first 25 years, computers were built using vacuum tube technology. This proved to be a barrier to the size and complexity of machines, since vacuum tubes are physically quite large (compared to a transistor), consume a relatively large amount of power, and produce a relatively significant amount of heat. Prior to the early 1960s, computers were extremely expensive and limited in the capability.

By 1965, commercial companies began to be built using solid state technology, greatly reducing their size, power consumption, and heat generation. By 1970, manufacturers learned to consolidate increasing amounts of electronic logic on micro chips, leading to small scale, medium scale, large scale, and finally *very large scale integration* (VLSI). Since the 1980s, chip designers have referred to chips with a high density of components as VLSI chips. This revolution in chip technology is the basis of a revolution in computers. The chip technology first boosted the mainframe computing technology, causing rapid development in large batch and timesharing computers. By the early 1980s, Digital Equipment had introduced the VAX computer, Apple was selling the Apple II, IBM had introduced the PC, and Sun Microcomputers was marketing their workstation. The VAX was wildly successful as a "departmental computer" that worked with (or in competition with) the enterprise's mainframe computers. Soon these departmental computers began to use local area networks (LANs) to connect with small computers (like PCs) and workstations, and ultimately to interconnect mainframes, departmental computers, workstations, and personal computers. Next, let's take a brief look at the hardware ancestors of SCCs.

[2] See http://www.opengroup.org/austin/. It is risky to include web URLs in an archival textbook, but in today's technology it is absolutely necessary. Please take any URL as no more than a hint that relevant content is published on the web. If the URL does not work, then use the context of the URL with your favorite search engine.

[3] See Chapter 2.1 of [Randell, 1973].

Creative Commons Attribution-Share Alike 3.0 U.S. License

- **Embedded Systems**. Engineers have used electronic circuits as parts of controllers for machines since the 1930s (slightly predating electronic stored program computers). In the 1970s, electrical engineers realized that it would be possible to simplify large circuit boards by replacing much of the hardwired logic with a programmable computer: The resulting device would be smaller, less expensive, and more flexible (since the software could be changed to reconfigure the logical circuit board). At about this same time, integrated circuit chip manufacturers began to supply *microprocessors* – tiny computers packaged on a single integrated circuit chip. Briefly, the first microcomputers were 4-bit microprocessors (like the Intel 4004). These machines were designed as miniaturized instances of conventional computers but with a 4-bit word rather than with a 32-bit or larger word like that used in the mainframe computers of the time. Like its larger counterparts, the 4-bit microprocessor CPU was designed to interact with a memory that contained a program and data, and to control input and output devices. The CPU instruction set, registers, and function units were designed to operate on 4-bit operands – the microprocessor could add, subtract, perform logical operations (such as OR and AND) on 4-bit quantities. (A 4-bit quantity can represent 16 different values, such as the positive integers from 0 to 15.) Since the CPU could handle 4-bit operands, it loaded and stored 4-bit words from/to the memory. A *microcomputer* could be built by combining a 4-bit microprocessor integrated circuit chip with memory and devices (sensors and actuators). Typically, such a microcomputer was built on a single circuit board with a microprocessor chip, memory, and a few specialized devices.
- **Personal Computing Machines**. By the late 1970s, Intel had upgraded the 4-bit microprocessor embedded systems microprocessor to an 8-bit model – the 8008. Not only did the 8-bit microprocessor provide the ability to convert larger amounts of hardware circuits to a computer, they could be used for primitive, but general purpose computing. An 8-bit microprocessors operates on 8-bit entities, each of which can take on any of 256 different values (the binary integers corresponding to the decimal numbers 0 to 255). These 256 bit patterns can also represent characters, for example using the ASCII coding standards; or they can represent small integers in the range of 0 to 255, or perhaps –128 to +127; and so on. Hobbyists seemed to be the first to recognize and exploit the power of 8-bit microprocessors and began to build their own "personal" 8-bit general purpose computers.

  The MITS Altair microcomputer kit enjoyed tremendous popularity with hobbyists, even though its only input device was a set of toggle switches and its output device was a set of light emitting diodes (LEDs) – in the first hobbyist machines there was no keyboard or display. Programs and data was entered into the Altair by setting 8 switches to represent a byte values, then by copying the switch settings into a byte in the memory. Output was presented to the user by displaying a byte value on a row of 8 LEDs – a bit value of zero was represented by turning the corresponding LED off, while a bit value of 1 was represented by turning the corresponding LED on. Just as the Altair microcomputer signaled the beginning of personal computing, it also started a new industry to provide software tools for personal computers. Some of Microsoft's earliest products were for the Altair microcomputer [Gates, 1996].

  The hobbyist market grew rapidly, so that by 1980, the Apple II and the IBM Personal Computer (PC) had been introduced as commercial products using a newer 16-bit microprocessor (such as the AMD 2901, the Motorola 6800, and the Intel 8088/8086). These newer machines were able to support far more sophisticated devices than toggle switches and LEDs of the Altair, including graphic displays, keyboards, and floppy disks. The 16-bit word also allowed the microcomputer to easily handle 8-bit character sets as well as 16-bit integers (with values ranging from –32768 to 32767). Meanwhile people had developed software library routines to treat a group of two or four 16-bit words as a single number – personal computers could be used as general-purpose machines.

  Through the 1980s, microprocessors continued to grow in word size (from 8 bits to 16 bits, then to 32 bits); by 1990 the Motorola 68000 chip family ("680x0") and Intel 80386/80486 ("80x86") microprocessors represented the state-of-the-art in 32-bit microprocessor technology. As software innovators created new products (called "killer applications") such as the electronic spreadsheet, businesses began to make heavy use personal computers. By 1990, personal computers had become an essential part of the business infrastructure, completely changing the environment from one of centralized mainframe computers and batch processing to a new world of PCs interacting with servers.

  Personal computers (IBM PCs and several others) stimulated the development of their own class of operating systems – the MacOS in Apple machines and MS-DOS in IBM-compatible PCs. The

Creative Commons Attribution-Share Alike 3.0 U.S. License

MacOS was designed especially to support robust graphics and user interface tools; MS-DOS focused on the file system. Early versions of both operating systems were intended to be used by a single person, running only one program at a time.

- **Workstations**. Another technology path split off from the hobbyist computer path. As 32-bit microprocessors began to appear in the 1980s, hardware designers realized that a microprocessor could be used to build a microcomputer that had as much raw computing power as modest mainframe computer (such as a DEC VAX or IBM System 38). These designers began to create single-board computers that incorporated the leading edge 32-bit microprocessors (such as the Motorola 680x0 or Intel 80x86) with memory, serial and parallel ports, and so on. These single-board computers were then used in high-powered single-user computers called *workstations*. A prominent example was the SUN (Stanford University Node) computer – which, of course, was commercialized into the first Sun workstations. A workstation was bigger and faster than a personal computer, yet small enough to be a single-user computer (contrasted with a timeshared computer such as the DEC VAX or a mainframe computer such as an IBM System 38 or 4300 series machines). Workstations were initially distinguished from personal computers by having 32-bit words rather than 16-bit words, and by operating at higher speeds. Secondly, workstations had much more complex operating systems than did personal computers – designed to support *multitasking* or *multiprogramming* – the computer could have several different programs loaded in memory, then switch the processor among these programs. In most cases the OS was a variant of the UNIX timesharing operating system.

  In the 1990s *reduced instruction set computers* (RISC) microprocessors began to replace the earlier 32-bit *complex instruction set computer* (CISC) microprocessors in workstations. Sun Microcomputers began to use their own Sparc 32-bit RISC microprocessor to replace the Sun 3 Motorola 680x0 microprocessors; HP used their own PA-RISC microprocessor, Digital Equipment produced the Alpha microprocessor, and so on. Intel upgraded the 80486 to the Pentium family of microprocessors, and began to phase out the 80486. At this time, Macintosh personal computers changed from the Motorola CISC microprocessor to the PowerPC RISC microprocessor, and machines that used Intel 80486 microprocessors with MS-DOS began to use the Intel Pentium microprocessor. As the Macintosh hardware changed, so to did the MacOS; similarly Pentium class machines began to use Microsoft Windows 95 or NT rather than MS-DOS. These new operating systems had the same kinds of capability as workstation operating systems: By the end of the 1990s there was no longer any significant difference between a personal computer and a workstation. The personal computer and workstation technology paths had converged to a single technology path.

While SCCs share fundamental ideas with all programmable computers, SCCs are most heavily influenced by embedded systems, personal computing machines, and workstations. Now we are ready to quickly review the general organization of SCC hardware.

1.2.1 The von Neumann Computer Architecture

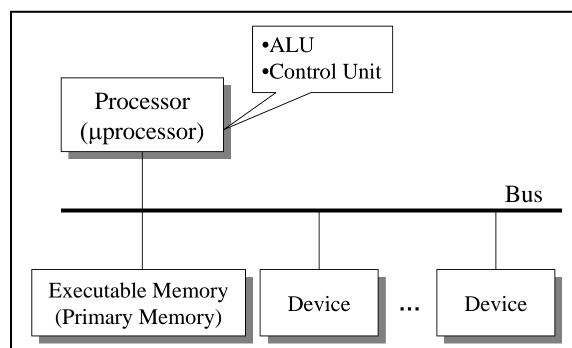


Figure 1-4: The von Neumann Architecture

Like the larger siblings, SCCs use the basic von Neumann architecture shown in Figure 1-4. The components are a processor (sometimes called a microprocessor in smaller machines), an executable (or

Creative Commons Attribution-Share Alike 3.0 U.S. License

primary) memory unit, and various devices, all interconnected by a bus. In the von Neumann architecture, the **bus** is a centralized switch that is used to transfer information among the other components.

1.2.2 The Processor

> The **(micro)processor** is also called the **central processing unit** (**CPU**). It contains the computational elements for a stored program computer, the arithmetic-logic unit and control unit.

The **arithmetic-logic unit** (**ALU**) performs arithmetic operations on computer words that contain binary encodings of data; for example, the ALU can compute the sum, difference, product, or dividend of two numbers stored in two computer words. The ALU also performs logic operations such as testing for zero, logical AND, and logical OR operations on computer words. The **control unit** manages the order of execution of instructions that are stored in the executable memory unit according to the computer's **fetch-execute cycle**. This little algorithm is the basis of the operation of all digital computers.

In order to understand how the fetch-execute cycle works, we can characterize its behavior using a pseudo code algorithm in Figure 1-5; ironically, we are using something that looks like software to describe how hardware behaves. In this figure, the **instruction counter** (**`IC`**) is a hardware register inside the control unit; it contains the executable memory address of the *next instruction to be executed*. The **instruction register** (**`IR`**) is another hardware register that always contains a copy of the machine instruction that is currently being decoded and executed. In the pseudo code description, the `haltFlag` represents a 1-bit hardware register that is set when the machine is to be halted. The control unit presumes that there is a machine language representation of the program stored in the executable memory. That is, a program has stored this program in the memory prior to directing the computer to execute it.

```
IC = address of the first instruction to be executed;
halt = FALSE;
while (haltFlag not TRUE) {
    IR = memory[IC];
    IC = IC + 1;
    execute(IR);
};
```

Figure 1-5: The Hardware Fetch-Execute Cycle

By studying the fetch-execute algorithm, you can see that the control unit is started by placing the address of the first instruction in a program in the IC and starting the fetch-execute cycle. The control unit will fetch the instruction from the executable memory location designated by the IC. Next, the control unit will execute this instruction by decoding the instruction and directing the relevant parts of the computer hardware (like the CPU) to perform a specific task (like add two numbers). After executing the instruction, the IC register has been incremented and the control unit fetches the next instruction, and so on. This design achieves the goal of enabling fixed electronic components to perform diverse tasks, simply by changing the program that is stored in the primary memory. This dynamic hardware control is unique to stored program computers.

1.2.3 Executable Memory

> The **executable memory** keeps a current copy of both the program and data during the time they are being operated on by the CPU.

Creative Commons Attribution-Share Alike 3.0 U.S. License

The executable memory is implemented by random acess memory (RAM) and various types of read-only memory (ROM). The RAM technology is designed to allow programs to copy information from the memory cells (a memory “read” operation) and to copy information into memory cells (a memory “write” operation). Most of the executable memory is configured as RAM, but small amounts are implemented using ROM – memory that is written once, but thereafter it is only read. The ROM memory is typically used to permanently store small amounts of information, usually programs, in the machine. Then, for example, when the computer is powered on, it can begin executing the program that is stored in the ROM part of the executable memory. Thus, the ROM programs are usually programs to get the computer up and running, and diagnostic programs to test the hardware for errors.

Modern computers represent information using the binary, or base 2, number system. Instead of having ten different digits as in the decimal, base 10, number system, binary numbers use only the binary integers (*bits*) 0 and 1. A binary number might look like 10010111_2 – it will never use the 2, 3, …, 9 digits. The binary number 10010111_2 has the same value as the decimal number 151_{10}. For our purposes, it is important to recognize that the hardware uses the base 2 number system, but the software incorporates tools that automatically convert base 2 numbers to base 10, and vice versa. Programmers and users usually need not be concerned with the fact that the hardware is using base 2 representations.

The executable memory is organized as a collection of individual cells, each with a numeric address (starting with zero) – see Figure 1-6. Executable memory manufacturers build RAM and ROM so that each cell contains a binary number with 8 bits, called a *byte*. The binary number 10010111_2 can be stored in one byte of the executable memory. The largest binary number that can be stored in a byte is 11111111_2, which is the same value as 255_{10}. Since general purpose computers often need to use much larger numbers, the computer can be designed so that it treats groups of 4 bytes as a single entity called a *word*. The largest number that can be stored in a word is over 4 billion in base 10 notation.

A computer’s executable memory hardware has three associated registers that are used to read and write the memory cell contents (either as bytes or as words):

- ***Memory address register*** (*MAR*): This register can be loaded with a memory address. If the executable memory had only 256 locations, then the MAR would only need to be large enough to hold 8 bits. The executable memories cells (bytes) would have the decimal addresses 0 to 255. Modern 32-bit computers can have an executable memory with up to 4 gigabytes of memory – a gigabyte is about a billion bytes (specifically, it is 2^{32} bytes).
- ***Memory data register*** (*MDR*): This register is intended to hold the contents of memory locations – usually a 32-bit (4-byte) word.
- ***Command register***: This register instructs the executable memory to perform some action – either a memory read or a memory write.

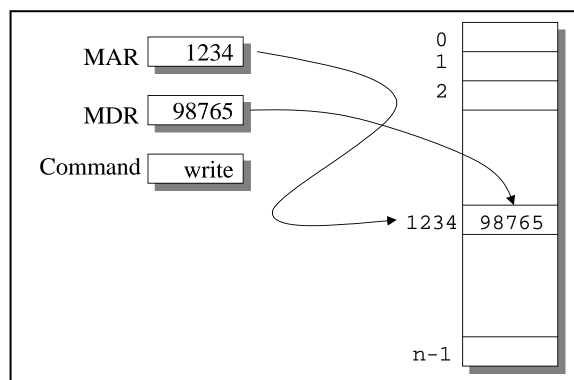


Figure 1-6: The Memory Organization

Creative Commons Attribution-Share Alike 3.0 U.S. License

To store information into a particular word or byte in the executable memory, the information is first copied into the MDR, next the address of the byte (or first byte in the word) is copied into the MAR, and finally the write command is copied into the command register. For example in Figure 1-6, the MDR contains the number 98765 and the MAR contains the address 1234 when the write command is copied into the command register, causing the value 98765 to be copied into the word that begins at address 1234. To read the value in a memory cell, first the target address is copied into the MAR, and then the read command is copied to the command register; after a predetermined amount of time, the contents of the specified address will be copied from the memory into the MDR.

1.2.4 Devices

The *devices* are parts of the computer used to introduce information into the computer, to report results that are stored in the executable memory, to store information when the computer is powered down, and to communicate with other computers. Examples of devices include a keyboard, mouse, display, floppy disk, hard disk, network interface card, sound card, mpEG decoder, and so on. *Input devices* (such as a keyboard, mouse, microphone or scanner) are used to transfer data from the external world into an internal memory unit. *Output devices* (such as a video display, speaker, or printer) are used to copy data from the executable memory into the external world. *Storage* devices (such as disk drives) are capable of storing bytes of information even after the computer is powered down – these kinds of devices are also called *persistent storage* devices. A storage device differs from the executable memory unit in that it is persistent; however the control unit cannot fetch instructions directly from a storage device. A *communication device* is an input and output device that transfers information into or out of an internal memory, usually via an external subcommunication network (such as the telephone system or a local area network).

All these components are designed to work together to "automatically" process information. External data is read with an input or network device, ending up in a storage device. When the executing program is ready to process the data, it first copies the data from the storage device into the executable memory unit. The CPU then processes the data a word at a time, writing the results back into the primary memory unit. The result data can then be exported with an output or network device, or saved for later use in a storage device.

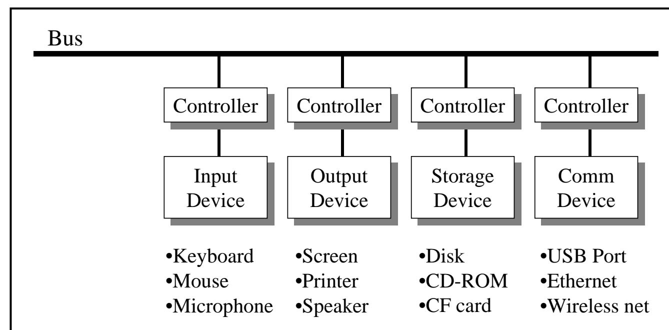


Figure 1-7: Devices and Controllers

Input/output (I/O) devices are attached to the computer bus (see Figure 1-7). An input device transfers data from a mechanism such as a keyboard, mouse, touch sensitive screen, or microphone into a CPU register. The CPU can then store the data into the executable memory. The CPU can fetch information from the executable memory into its registers, then write the information over the bus to an output device such as a computer screen, a speaker, or printer. Communication devices can transmit information to another location. For example serial and parallel ports, infrared transmitter/receivers, wireless network cards, and network interface cards are all communication devices. Storage devices can also be used for input and output: The input operation causes data to be moved from the storage device (such as a magnetic or optical disk) to the CPU registers, then into the primary memory. The output operation moves data from the primary memory to the storage device.

Creative Commons Attribution-Share Alike 3.0 U.S. License

The specifics of the device operation depend on whether it is an input, output, communication, or storage device, and on the way the specific device works (for example, reading input information from a mouse is different from reading a keystroke from a keyboard device). There are many types of devices of each of the four classes, ranging from slow and inexpensive ones to fast and expensive ones. As a result, there is a wide range of interfaces to the device controllers that the OS must be able to manipulate properly in order to correctly operate the device.

Each device has a specialized device controller, which adapts that particular device type to the particular type of computer bus. For example, if you wanted to attach a SCSI disk to a PCI bus, then a PCI-SCSI controller would be used to adapt a disk's SCSI interface to the PCI bus. Besides the physical adaptation, the controller's job is to adapt the SCSI hardware signals to the PCI bus, so that the CPU can send electronic signals to the device. Another aspect of the controller is to translate a standard set of commands (from the CPU) into a set of commands that are specific to the device. This is necessary because each device has it own peculiar set of commands, but the controller can translate the CPU's generic device commands into the specific commands needed to operate that particular device. Conversely, the controller also translates information that the device transmits to the computer, for example error notifications. One other important aspect of the controller's job is to provide continuous attention to the device. This is necessary since the device may required immediate (and unpredicted) attention because of some unusual condition – ranging from simply repeating a failed read/write attempt, to gracefully powering the device down because of too much heat.

Device controllers for different hardware device types are unable to present exactly the same interface to be used by the CPU, but they can make the CPU-device interface be much more similar than if there were no controller. Modern computers depend on the presence of controllers for this logical translation as well as for the physical adaptation of the device to the bus.

The details of the physical device interface depend on the particular computer, bus, and device types; however, we can learn the general concepts for how this interface can be designed by considering a hypothetical interface like the one shown in Figure 1-8. The controller interface must support interaction with software executing on the CPU in such a manner that it will be possible for the software to determine the state of the controller (and device), to direct the controller to execute chosen commands, and for data to be transferred to/from the device from/to the CPU. In the figure, the controller incorporates a set of three registers that can be read or written by the CPU:

- ***Command register***: The CPU writes a device-specific command into this register whenever it wants to direct the register to perform an operation. For example, the command might be to read data from the device, or to turn the power off from the device.
- ***Status register***: The controller posts information in the status register so that the CPU can read it to determine the internal state of the device. For example, the device might be idle, busy, or have experienced an error condition while executing the previsous command.

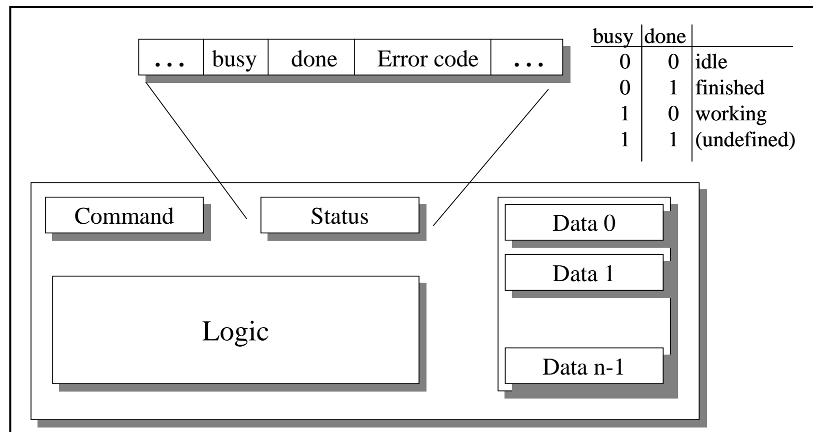


Figure 1-8: The Conceptual Device Controller Interface

Creative Commons Attribution-Share Alike 3.0 U.S. License

- ***Data register*(*s*)**: Prior to performing a device write command, the CPU can place data into one or more of the data registers; the write command will then copy the data from these data registers into the device. For example, a text printer device's data registers would be loaded with the next character (or characters) to be printed by a write command. The data register(s) are also used for input, or read, commands. The device responds to a read command by retrieving data from the input device and storing it in one or more data registers.

The figure indicates that the status register has various fields in it, for example the error code field is set to indicate errors that might have occurred on the last operation. There are also two 1-bit fields called the *busy* and *done* flags. These are used for the CPU and the device to coordinate their operation with one another. This is done by letting specific settings have meaning with respect to the synchronization state of the device:

- busy = 0 and done = 0: In our conceptual model, this state means that the device has completed its previous I/O operation and can accept a new command.
- busy = 1 and done = 0: This configuration of the two 1-bit fields means that the CPU has issued an I/O command (by writing it into the command register), causing the controller to immediately set the busy flag. The device is busy working on the current command when the two flags are in this configuration.
- busy = 0 and done = 1: This controller changes the 1-bit flags to this configuration after it has completed the I/O operations. It sets done to 1 to signal the CPU that it has completed the operation, rather than to the idle state (busy = 0 and done = 0) so that the CPU will have a chance to read the status to determine the error code, and to clean up after the operation completion if it wants to do so. For example, after an input operation, the CPU would need to copy the information from the data register(s) into the CPU registers or the executable memory before the device was made idle in preparation for the next operation. After the CPU has finished the clean up, it will write a clear operation into the command register, and the controller will set done = 0.

This use of the busy and done flags is necessary because each device can physically be in operation at the same time as the CPU is executing instructions. The flags make it possible for the (software on the) CPU to explicitly check the device controller to determine when the corresponding device has completed an operation. OS people like to describe this simple act as one of *synchronizing* the CPU and the device operation, since the CPU will execute one instruction sequence if the device is still busy (such as check the device status again), and it will execute a different instruction sequence if the device has completed (such as continue the execution of the application program).

In the simplest of devices there is only a single data register in the controller. For example a serial port device typically has only a single byte data register. However, some devices are constructed so they read many bytes at once, for example a disk driver; other devices can read many characters on a single input operation, and will put each of the characters in a separate data register in the controller. So whether or not the controller has multiple data registers depends on the design of the device. In those cases where the controller has enough data registers to store the result of several read operations, we say that the device controller contains an input buffer – implemented as an array of data registers.

Interrupt-driven I/O

As you have just read, the CPU communicates with a device by transmitting commands, status, and data back and forth over the bus. For example, if a program were written to read information from a keyboard device, the CPU performs the following steps:

1. Check the status of the device to see if it is idle or not. (If it is not, then continuously check the status of the device until it to becomes available; this is called *polling the device*.)
2. Send the read command to the keyboard device.
3. Repeatedly check the status of (poll) the device to determine when it has completed the read operation.
4. Transfer the information from the device input buffer into the primary memory.

You can see that for keyboard input operations, the CPU could waste a lot of time polling the device in Step 3 while waiting for the user to press a key on the keyboard.

Creative Commons Attribution-Share Alike 3.0 U.S. License

In the 1960s the basic von Neumann architecture was improved by providing a new approach for device I/O: The CPU starts the device, then ignores it until the device *explicitly notifies* the CPU that it has completed executing the command. How is this done? First, a 1-bit **interrupt request register** (which we will call `InterruptRequest`), is incorporated into the CPU hardware. Next the control unit is modified so that it checks this register's value during each instruction fetch-execute cycle (see Figure 1-9). Conceptually, the hardware connects all device status flags to the interrupt request flag in the control unit using inclusive-OR logic. Whenever the I/O completion status flag is set in any device, the `InterruptRequest` register is set in the control unit. The control unit will be aware of a device completion within the time it takes to execute a single instruction – generally within nanoseconds (depending on the time taken to execute the current instruction).

This signal from the device hardware to the control unit – the **interrupt** – causes the processor to cease executing the sequence of instructions addressed by the PC and to branch to a new instruction sequence whose address is stored in, for exaPBle, memory location 1 (denoted memory[1] in the figure).

When the branch occurs, the hardware saves the old IC contents (address of the next instruction to be executed) of the interrupted program execution. The address of the interrupt handler is stored as an indirect address in memory[1] when the machine is started. The **interrupt handler** is a part of the OS that will be executed when any device completes its operation. With this interrupt mechanism, the application software does not need to continuously poll the device to detect when it has completed.

```
while (haltFlag not FALSE) {
 IR = memory[IC];
 IC = IC + 1;
 execute(IR);
 if (InterruptRequest) {/* Interrupt the current process */
  memory[0] = IC;    /* Save the current IC in address 0  */
  IC = memory[1];    /* Branch indirect through address 1 */
 }
```

Figure 1-9: The Fetch-Execute Cycle with an Interrupt

Interrupts are an important addition to the von Neumann hardware. Besides saving large amounts of CPU time (by avoiding polling), they make it possible to design an entirely new style of operating systems. As you will see in Part 3, machines with interrupts can *guarantee* that the OS will execute periodically. Therefore we distinguish between operating systems for hardware that do not use interrupts and those that do.

Some Examples of SCC Devices

Specialized devices for SCCs are miniaturized to be compatible with small computers, and usually have low power consumption so that they will not use too much of the battery power of mobile computers.

PCMCIA. The Personal Computer Memory Card International Association (**PCMCIA**) established one of the early (circa 1989) standards for devices for small computers. Originally conceived as a memory card that could be added to an SCC, the standard defined a form factor (about 2" wide, 0.15" high, and 3.25" to 5" in length), a 68-pin connector on one end, and a protocol for the signals on the pins on the connector. Any manufacturer could then create a card that would fit into a PCMCIA slot (or bay) on an SCC, and any SCC manufacturer could provide a PCMCIA slot to accommodate a memory card. This enabled the end user to add memory to an SCC by buying a PCMCIA memory card and plugging it into the PCMCIA slot.

As PCMCIA increased in popularity, manufacturers realized that they could create PCMCIA compatible cards that provided other services besides memory, for example 3COM, US Robotics, Xircom,

Creative Commons Attribution-Share Alike 3.0 U.S. License

and other manufacturers created telephone modem and Ethernet PCMCIA cards. One could buy such a card, insert it into the PCMCIA slot, install appropriate OS components, then have a communication device on the otherwise isolated computer. Today, most add-on (not built into the computer) wireless devices are packaged as PCMCIA cards. Today, PCMCIA cards cover a diverse set of services ranging from extended primary memory, to communication devices, to secondary storage devices, even including GPS devices.

Other Storage Devices. As PCMCIA led the way to standardized devices for SCCs, even smaller devices became available to provide storage. The leader among the early devices was the Compact Flash (CF) storage card (physically, about 1.5” × 0.13” × 1.4”) to provide memory. Again, computer manufacturers could then add a Compact Flash bay to accommodate this type of memory. Compact Flash memory did not have much impact on the notebook computers, but it was heavily used on smaller computers such as PDAs. As consumer electronics began to use digital logic, Compact Flash cards became an essential element of their operation. For example, they are often used to store images on digital cameras.

Compact Flash was the earliest such device to market, but it was followed by various other kinds of memory, including the SmartMedia card, Sony memory stick, the Secure Digital (SD) card, USB memory devices, and others. Consumer electronic products may use any of these standards for memory devices, hence SCCs may use any of them, although the CF and SD cards seem to dominate in today’s SCCs.

Specialized Graphics and Video Devices. Many SCCs use a specialized graphics or video device to support the human-computer interface (HCI). For example, a cell phone may use a light emitting diode (LED) display rather than a phosphorescent screen such as those on a conventional desktop. There is an emerging technology for small video screens, led by their use in digital and video camera displays. Cell phones that transmit and receive photographs have similar requirements for inexpensive, low power graphic devices.

Tablet/touch devices. PDAs led the move toward computers that use a stylus as the HCI input device (replacing the conventional keyboard and mouse). Their popularity in PDAs has led to the introduction of tablet computers – notebook style computers that have a stylus-sensitive screen; the user can employ a conventional notebook keyboard and pointing device, or use the stylus.

Audio Devices. Desktop computers often incorporate microphones and speakers. However, the microphones are largely ignored (the speakers are heavily used to play back mp3 files). In PDAs, the microphone and speaker is more heavily used, for example, to record voice notes and annotations. It is this use of PDAs that leads to the now common cell phone that incorporates conventional PDA features.

Sensors and Actuators. Embedded system devices are often sensors and actuators: A **sensor** is a device that reads an external mechanism such as a thermometer, and then digitizes (if necessary) and transmits the thermometer value as input data to the SCC. An **actuator** is an output device that controls an external mechanism. For example an SCC embedded in a robot will have a series of sensors and actuators to control the robots arms, motor, wheels, and so on.

1.2.5 Privileged Instructions and the CPU Mode Bit

Some CPUs incorporate a 1-bit mode register to define the execution capability of a program on the CPU:

> The **mode** bit can be set to **supervisor** mode or **user** mode. When the CPU is in supervisor mode, it is enabled to execute every instruction in its hardware repertoire. When the CPU is in **user** mode it is only allowed to execute a *subset* of the instruction set. Instructions that can be executed only in supervisor mode are called **supervisor**, **privileged**, or **protected** instructions to distinguish them from the user mode instructions.

Creative Commons Attribution-Share Alike 3.0 U.S. License

I/O instructions are privileged instructions, so when a program executes in user mode, it cannot perform its own I/O. Instead, it must request that the OS (which executes in supervisor mode) perform the I/O operation in its behalf. This is done by having the user mode program make an OS call using a special hardware instruction that switches the CPU to supervisor mode and begins to execute the OS software (which then executes the actual privileged I/O instructions).

Privileged instructions prevent indiscriminate operations on shared hardware resources. For example, a disk drive usually contains many different users' files; if any program could read or write the disk, then an unauthorized user could read or write any other users' files. The mode bit also enables the OS to provide a *protection barrier* between itself and the application programs, and between any two executing application programs. As you will see in Part 4, general purpose operating systems rely heavily on interrupts and the mode bit to guarantee that rogue software does not access information or execute instructions for which it is unauthorized to do so.

1.2.6 System-on-a-Chip Technology

The evolution of SCCs was fueled by hardware innovation, including very large scale integration at the chip level. In the 1990s, chip technology had become so refined that computer architects realized that they could incorporate certain device functionality on the same chip that implemented the CPU. The first extensions were cache memory to reduce contention for the shared bus. Then designers began to experiment with adding device and bus interface technology to the CPU chip. This was the beginning of the **system-on-a-chip** (**SoC**).

An early motivation for SoC came from PDA like devices that needed to miniaturize the size of the CPU and a graphics functions. A specialized graphics chip can perform a number of functions ranging from rapid bit transfer (called "bitblit") from one memory to another, to hardware image rotation and scaling. In the 1990s chip manufacturers created products to accelerate graphic operations, and then sold them as separate chip sets. An SCC SoC could be designed so that this graphics engine and the CPU were packaged on the same chip.

System-on-a-chip has been used as an advertising buzzword to describe many different microprocessor chips. The Transmeta Crusoe chip illustrates substantial SoC technology that is well suited to SCCs: The supplementary functionality added to the CPU include three memory interface controllers and a PCI bus controller. This reduces the number of chips required to interconnect memories and the bus with the CPU chip.

1.3 Communication Technology

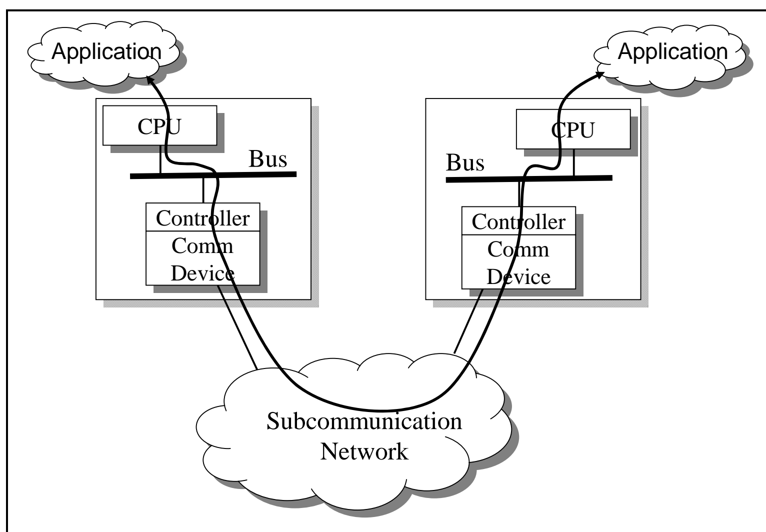


Figure 1-10: Network Communication

Creative Commons Attribution-Share Alike 3.0 U.S. License

Networks are not considered to be an essential component of every von Neumann computer; in fact standalone computers do not even use a network. However, a significant aspect of contemporary small computers is their ability to communicate with other computers using a network. The implication is that every SCC contains at least one **communication device** that enables it to exchange information with a remote computer. The idea, illustrated in Figure 1-10, is that each computer (such as an SCC or a server machine) contains a device that can use a subcommunication network to transfer information from one computer's communication device to another computer's device. The subcommunication network could be the public switched telephone network, a specialized network (such as Ethernet) especially intended for data communication, or the public Internet. Once a computer has this communication ability, the SCC's OS and middleware can create various abstractions that allow the applications to exchange information such as files, email, or messages. This ability to execute code within a machine and to exchange information with other computers is the foundation of distributed computing.

1.3.1 Communication Devices and Networks

In the 1980s, **local area network** (**LAN**) exploded onto the technology scene.

> A **LAN** is a digital subcommunication network that allows a collection of computers in the same physical proximity to exchange information over a shared medium.

The Ethernet and IBM Token Ring LANs were both introduced as commercial products in about 1980, and both quickly became popular to interconnect personal computers, workstations, departmental computers, and mainframes. The primary early use of the LAN was as a means to transfer files and electronic mail among the machines, but later the uses became more sophisticated with remote file systems, remote procedure call, remote objects, and other distributed system mechanisms.

In about 2000, **wireless networks** became cost-effective for LAN implementations. Many vendors sell components for an IEEE 802.11a/b/g (also known as **WiFi**) wireless LAN that can transmit information within a sphere of radius ~500 feet (different versions operate at rates up to 54 Mbps). The IEEE 802.15 (Bluetooth) wireless personal area network (**WPAN**) is another kind of network that covers a 10 meter sphere with a data rate of 1 Mbps – but using much less power than an 802.11 wireless transmitter/receiver. Bluetooth is probably best known today as being the wireless technology to connect a microphone and earpiece to a cellular phone. Cellular telephone technology offers wireless service for longer distances, but usually at a lower transmission rate.

Today, there is a logical network of networks, meaning that it is possible to interconnect separate LANs – wired and wireless – together so that a computer connected to one LAN can exchange information with a computer that is on a completely distinct LAN. In short, the way this is accomplished is by having at least one computer on each LAN that is also connected to another LAN. This **gateway** computer has software that forwards information that it receives from a host computer on one network to a host computer on a different network. This is the basic idea behind the public Internet – thousands of LANs are interconnected by gateways that forward information across other LANs to remote hosts.

1.3.2 Network Protocols

There are many varieties of network devices, each designed to use its own method for sending and receiving information. For example, an Ethernet controller looks quite different from an Asynchronous Transfer Model (ATM) network or a wireless 802.11b net. One of the major challenges in network-based computing is to ensure that a computer with a network device will have software to control that device so that is able to exchange information with software on a remote computer where the remote software is controlling the behavior of the network device on its machine. In short, this requires that the two programs be designed so that they coordinated actions, for example actions that cause one computer to listen while the other is transmitting information.

Network designers have built approaches that enable a program on one computer to direct its network device so that it cooperates with the activity of the network device on a different computer. This is done by defining a set of communication protocols for both parties. A low level communication protocol specifies things like when one device should transmit and when it should receive. A high level protocol might

Creative Commons Attribution-Share Alike 3.0 U.S. License

specify things like the format of a floating point number. The International Standards Organization (ISO) **Open Systems Interconnect** (OSI) system architecture model is a widely agreed upon collection of such protocols – all contemporary networks use the OSI model at least to some degree. We will look more carefully at this OSI model for communication in Chapter 2. For now we note that machines in a network can use different processors and different operating systems, yet still able to exchange information provided they use consistent ISO OSI protocols.

> A **network protocol** defines a common set of data formats and behavior for computers that wish to communicate with one another.

This information exchange among *heterogeneous* systems is possible because the different computers use common network protocols.

1.4 SCC Operating Systems

An OS is the software that directly controls a computer's hardware, while exporting an API that simplifies the use of that hardware. In OS circles, we think of the OS requirements problem as being one of identifying an idealized virtual machine, then creating a collection of functions (advertised on the API) to implement that virtual machine on top of the hardware. Device abstraction is a natural first-step in the evolution of virtual machine technology. The earliest operating systems focused on device abstraction much as was done in the Basic Input/Output System (BIOS) on an IBM-style personal computer some 35 years later.[4]

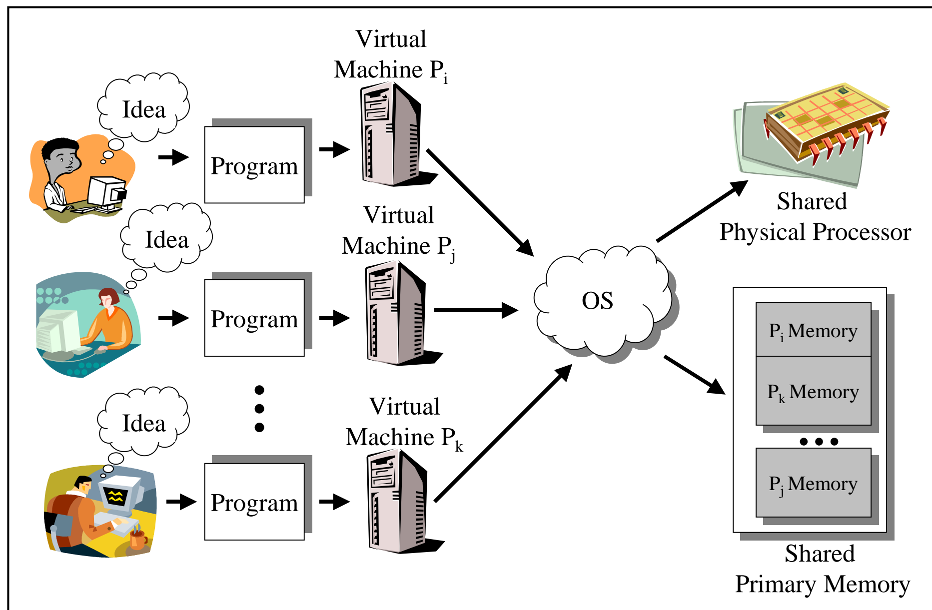


Figure 1-11: Abstracting the CPU and Memory

This virtualization of devices is the cornerstone of operating systems. As operating systems evolved, the sophistication of the virtual machine abstraction evolved to virtualize the CPU and executable memory. In Figure 1-3 you saw 3 different levels of virtualization of a bitmap display device. Somehow, we need to create a virtual CPU and executable memory if we intend to abstract the operation of that part of the hardware. Roughly speaking we will need the OS to create a computing environment that has the properties illustrated in Figure 1-11. In this approach *each executing program is written as if it had its own machine on which to execute*. The OS exports this abstraction by managing the way that the CPU and

[4] [Messmer, 1995] is a very comprehensive discussion of PC hardware.

Creative Commons Attribution-Share Alike 3.0 U.S. License

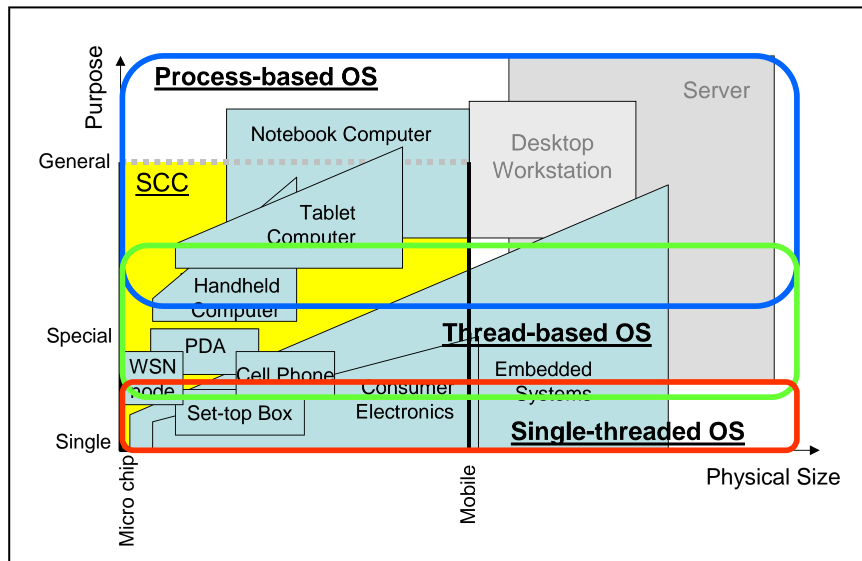
primary memory are shared among the executing programs (processes). That is, the OS exports a virtual machine for each executing program. Ideally, the virtual machine appears to be a physical machine with appropriate individual component abstractions (like an easy-to-use display device interface).

A **multiprogrammed** OS exports *multiple virtual machines* that can exist at the same time. Informally, modern operating systems implement multiprogramming by allowing multiple programs to be simultaneously loaded into different parts of the computer's physical executable memory. Next, the OS keeps a record of the logical status of each virtual machine – for example, is the virtual machine is running, or is it waiting for some device to complete an I/O operation? The set of all virtual machines that are logically running have a program loaded in their region of the memory, and would use the computer's processor if it were available. If there are *N* different virtual machines logically running but only one physical CPU in the computer, then at any given moment, the OS will choose one of the logically running virtual machines to use the physical processor. Later, it will switch the processor so that the currently-running virtual machine is suspended (but still logically running), and a different virtual machine is allowed to use the physical processor.

The classic requirement for an OS is to create appropriate abstractions of the hardware operation for a given software domain. An SCC OS is responsible for these same tasks, though the software domains are a little different from conventional desktop and server machines. Specifically, important differences are that:

- SCC devices are different from conventional machine devices, so some of the hardware abstractions for an SCC are different from those used in conventional machines.
- An SCC is configured with fewer resources than a desktop or server machine, so the OS implementations (abstractions) are more modest than their counterparts in large systems.
- A family of similar SCCs can use the same basic OS, but the resource constraint may require that different members of the family use different virtual machine policies, depending on the specifics of the physical SCC. The OS should support flexible configuration and policies.
- SCCs often appear as some form of embedded system. For example, the SCC might be a player for streaming media being delivered over the network connection. In contrast to traditional operating systems, this class of operating systems frequently uses real-time techniques for managing the behavior of streaming media resources.
- Since SCCs are the foundation of new computing environments such as internet appliances, it is likely that the OS will need to adapt to different computing paradigms such as rapid evolution toward web browsers as OS and interpreted execution (as in the Java Virtual Machine).

One of the interesting aspects of SCC operating systems is their diversity. Systems that focus on a single task need not have an OS that was designed to handle 100 timesharing users. We can systematically consider OS technology by considering 3 general classes of operating systems corresponding (suggested by Figure 1-12):



Creative Commons Attribution-Share Alike 3.0 U.S. License

Figure 1-12: Classes of SCC Operating Systems

- **Single threaded (ST) OS**. These operating systems are designed to manage SCCs that have a small range of types of work to perform. For example, the SCC might only be responsible for managing a sprinkler system. The hardware required to support this type of OS does not need to be as sophisticated as it would be for computers that have to handle more diverse work, so the OS can be relatively simple. Historically, there is a class of operating systems – often used in embedded systems – that address this need without using all the features of modern computers. Although these systems take full advantage of the von Neumann computer architecture, they do not rely on the existence of interrupt-style devices nor a CPU with a mode bit. As a consequence, these systems can be built with very simple software and use very inexpensive microchips and architectures; however they do not support general purpose computing at all.
- **Multi Threaded (MT) OS**. If the system hardware supports devices with interrupts, it is possible to implement an OS that provides more comprehensive, multitasking support than is possible in the ST OS. When the hardware incorporates a timer device with an interrupt, the OS can be assured of gaining control of the machine periodically, whereas that is not possible in machines without interrupts. This means that interrupt-driven operating systems can be used to execute software for more general computing problems than can single purpose systems (again consider Figure 1-12). Nevertheless, since the CPU does not contain a mode bit, it is not possible to ensure that one executing program does not interfere with the execution of other programs. This class of operating systems first implemented multiprogramming in the 1960s and 1970s. While those systems referred to the schedulable unit of computation as a *process*, since there are few/no protection barriers between the units of computation today we would generally refer to such units of computation as "threads." Consistent with this modern perspective, we say these operating systems are thread-based. We will focus on this MT operating systems in Part 3.
- **Multi process (MP) OS**. By the late 1980s, the classic process idea had evolved into a robust virtual machine model where activity within each *process* was assured of isolation from other processes, and each could support multiple threads of execution. The hardware for these systems incorporate devices with interrupts and a CPU with a mode bit. By exploiting the mode bit, the OS can provide very effective security barriers between process execution environments, preventing threads from unauthorized access to information belonging to another program or user. Computers with MP operating systems can safely be used for the full spectrum of information processing tasks, ranging from real-time control to database management systems. Windows XP (as well as the predecessor NT and 2000) and the UNIX family of operating systems are TB operating systems.

This characterization provides a general framework for studying operating systems. It allows us to learn about the parts of OS technology that do not rely on the presence of interrupts, and to see how these technologies can be used to manage SCCs that are either single-purpose of special-purpose systems. In Part 3 we consider the kinds of operating systems that are built to exploit hardware with interrupts. This type of OS has often been used on special-purpose SCCs (but they can also be used with single-purpose systems). In Part 4 we consider the PB operating systems that are built with modern hardware facilities. It is important to remember that as integrated circuit technology has evolved, interrupts and a mode bit can be incorporated into very inexpensive CPU microprocessor chips. This implies that increasingly, it is possible to use a PB OS on a single-purpose system. However, by doing so, the resulting software execution environment may be overkill for the application domain. Production quality SCCs often employ general hardware, but with limited OS capability in order to simplify their software development environment.

An SCC is generally not assumed to be "large enough" to store all the applications and data that it will ever need. Instead, the SCC is expected to use a network to exchange information with one or more remote computers, generally called **server** machines. Operating systems began to focus on distributed computation in the 1980s – primarily driven by the integration of network and workstation technology – hence they were optimized for workstation environments and 10 Mbps LANs. SCCs employ contemporary high speed, low power consumption 32-bit microprocessors, wireless networks, and (sometimes) multiprogrammed OS environments. (Desktops and servers are not concerned with power consumption, so their microprocessors can generally operate at a higher cycle rate than an SCC's microprocessor.) SCC application software will share local and remote information at varying granularity (files, messages, objects

Creative Commons Attribution-Share Alike 3.0 U.S. License

and other data structures, or bytes). Several SCC application domains require support for streaming media data types – audio and video streams. In combination, the environment has stimulated an evolution in OS technology. Dertouzos and Gates hypothesize a full revolution in such operating systems: "Browsers and operating systems will merge in functionality, simply because people need to have the same commands for dealing with information, regardless of whether it is local or distant."[5]

1.5 Summary

SCCs have emerged as a viable technology, encouraged by the interest in consumer electronics, the popularity of the Internet, and radical innovation in computer hardware miniaturization. SCCs include embedded systems, set-top boxes, cell phones, WSN nodes, PDAs, handheld computers, tablet computers, and small notebook computers.

Single-purpose computers, especially computers for embedded system applications, can operate without using device interrupts of a CPU mode bit. This greatly simplifies the requirements on the OS, enabling designers to design the entire software system as a single execution of a single program. This approach tends to minimize the investment in software.

Multi threaded computers – computers that support the execution of more than one thread at a time – depend on device interrupts, particularly a timer device with interrupts, in order to assure that the OS correctly implements CPU sharing among the threads. The timer interrupt occurs periodically, and each time it occurs, the OS has an opportunity to execute. It can thereby multiplex the CPU from one thread to another.

Process-based computers depend on the presence of a CPU mode bit to distinguish privileged and user instructions. By creating this distinction, a computer with a process-based OS can ensure that one process is, for example, unable to read or write parts of the primary memory that have not been allocated to them.

SCCs are distinguished from desktop computers by the *requirement* that they include some type of networking capability. The rationale is that an SCC will not necessarily be able to incorporate all of the hardware it will need for storage or peak computation; it will rely on the communication network to retrieve information, and to request specialized computation as needed. This means that SCCs necessarily rely on distributed software for correct operation – a decided difference from typical desktop OS requirements.

1.6 References

1. Dertouzos, Michael L. and Bill Gates, "Titans Talk Tech: Bill G. and Michael D.," MIT's Magazine of Innovation Technology Review, May/June 1999 (available at www.techreview.com/articles/may99/qa.htm).
2. Furht, Borko, Deven Kalra, Frederick L. Kitson, Arturo A. Rodriguez, and William E. Wall, "Design Issues for Interactive Television Systems," IEEE Computer, 28, 5 (May 1995), pp. 25-39.
3. Messmer, Hans-Peter. *The Indispensable PC Hardware Book,* 2nd ed., Reading, MA: Addison-Wesley, 1995.
4. Randell, Brian, editor, *The Origins of Digital Computers: Selected Papers*, Springer-Verlag, Berlin, 1973.

[5] See www.techreview.com/articles/may99/qa.htm.

Creative Commons Attribution-Share Alike 3.0 U.S. License