LABORATORY INTERFACING HANDBOOK

digital™

LABORATORY INTERFACING HANDBOOK

Prepared by
Laboratory Data Products Group

1st edition, September 1986
2nd edition, December 1986

The postpaid USER'S COMMENTS form on the last page of this document requests the user's critical evaluation to assist in preparing future documentation.

This document was prepared using an in-house documentation production system. All page composition and make-up was performed by TEX, the typesetting system developed by Donald E. Knuth at Stanford University. TEX is a registered trademark of the American Mathematical Society.

Contents

Chapter 2 Realtime I/O Modules

Chapter 3 How To Perform Analog Input

Chapter 4 How To Perform Analog Output

Chapter 5 How To Perform Digital I/O

Chapter 6 How To Use The IEEE 488 Instrument Bus

Chapter 7 How To Control Serial Devices

Chapter 8 How To Use A Clock/Counter Module

Chapter 9 Advanced Realtime Programming Techniques

Chapter 10 Realtime System Performance

Figures

Tables

Preface

LABORATORY INTERFACING HANDBOOK

In the last five years, computers have made an important transition in the laboratory environment. Prior to that period, they were an esoteric, infrequently-used, highly specialized tool for selected lab applications. Today computers are a standard, almost universally accepted part of lab instrumentation. The proliferation of personal computers and of small, specialized vendors of data acquisition software and peripherals has made it possible to buy and use a "turnkey" system for many applications without the need for any computer programming and with almost no knowledge of device interfacing beyond how to plug two cables together.

This turnkey approach has obvious advantages for individuals doing routine repetitive testing who lack the background knowledge to interface their computers to lab devices. However, it can have major disadvantages for the research scientist who must adapt to frequently changing application requirements, and for whom a black-box approach to experimentation is counter-productive in the long run. A turnkey approach can also create serious problems for the volume-testing laboratory manager who finds many expensive and often incompatible turnkey systems being purchased where a single general-purpose system could be programmed to solve the same combination of problems much more efficiently.

The Laboratory Interfacing Handbook is intended to help scientists evaluate such tradeoffs by providing practical knowledge on the general problem of interfacing computers to laboratory devices. It is aimed at:

- Novices who want to understand their turnkey systems in greater depth
- Beginning programmers who want to modify turnkey software or start developing their own
- Advanced programmers who have no experience with realtime devices or software

For the computer novice, the Handbook is designed to be read in sequence and in detail. Those who have some knowledge of instrumentation, signals, and interfaces can skim or skip Chapter 1; those with some knowledge of computers and I/O hardware can skim or skip Chapter 2. Chapters 3-8 contain detailed information by type of application, and can be read individually or in any order.

The emphasis of the Handbook is on problem-solving. There are ample pre-existing references (see the Bibliography) on the technology and theory of data acquisition and control systems. The focus here is on the intelligent application of these systems.

Chapter 1

Introduction To Realtime Computing In The Lab

When a computer is interfaced to laboratory devices, it performs time-critical operations such as varying control settings during an ongoing experiment in which test data are captured as a function of time. In computer science, such operations are known as "realtime" applications because the computer must respond to external events within a known time interval. While time deadlines with human users are typically flexible, the deadlines are fixed in realtime applications.

The term *realtime* does not in itself imply high performance; realtime applications frequently are quite modest in their demands on a computer system. Whatever work is to be done, however, must be accomplished within some time limit, or the application will fail.

Realtime operations are of two main types: data acquisition and control. Often both are required in a single application.

Data acquisition involves capturing experimental or test data for subsequent analysis, display, or storage. Typically, each data value must be labeled implicitly or explicitly with the precise time at which it was captured; such "time-based" data acquisition can be initiated by the user's program or by an external trigger event. The experiment or test itself can be controlled by a human operator or by the computer.

Control operations involve not only data capture, but also information output to manage one or more aspects of the external device's operation. In many laboratory situations, the control requirements are limited to synchronizing the computer and instrument operations by recognizing when one test sequence is over and signalling the instrument to begin another. This is essentially an *open-loop control* requirement, since the computer does not affect each test sequence while it is in progress.

In some situations, however, the computer uses inputs to calculate control outputs that keep the test sequence within predefined limits. In such *closed-loop control* applications, the entire three-step sequence of acquire-process-output must be performed within the relevant time constraints of the application. Closed-loop control applications are frequently found in manufacturing and process control applications, as well as in the scientific laboratory.

1.1 Forms And Sources Of Realtime Data

Hundreds of different types of lab devices can be interfaced to computers, but these devices generate only three main forms of data: analog, discrete digital, and time-interval. Understanding these forms of data is the first step in solving the overall lab interfacing problem.

1.1.1 Analog Signals

In scientific research, the data to be captured are most often changes induced in physical parameters that must be measured during the course of an experiment. Variables such as temperature, weight, pressure, speed, acceleration, transmittance, and absorbance are the actual parameters of interest. However, such parameters must be measured using a device which converts the value of the parameter at each point in time to a corresponding electrical signal. Such devices are called *transducers*. The resulting electrical signal is an "analog" of the underlying parameter, and quantitative measures of the analog signal can be converted to quantitative measures of the parameter at corresponding points in time.

Temperature is one common parameter which is measured in this fashion, and the thermocouple is the most frequently used type of temperature transducer. In a thermocouple, two pieces of dissimilar metal placed in isothermal contact exhibit a potential difference as a function of temperature. In general, these voltages are small and depend on the metals employed.

When a thermocouple is used to measure temperature in an experiment, the result is a continuously varying analog signal. If the thermocouple output is nearly proportional over the temperature range encountered, these voltage values can be converted to analogous temperature values using a linear transformation. (A nonlinear relationship simply involves a slightly more complex mathematical transformation).

Analog signals are also used as computer outputs to control experiments. In such instances, the signal is an analog to a control parameter, such as flow, pressure, or acceleration. In cyclic voltammetry, for example, a triangular voltage waveform is used to induce oxidation or reduction of electroactive substances. These reactions are measured via the electric current they generate. A lab computer can be used both to generate the required voltage scans and to acquire the reaction data. Plots can be generated for qualitative analysis, or the data can be reduced for quantitative determination of peak amplitude, wave slope, reversibility, etc.

Whether analog signals are inputs or outputs to the computer, they require special data-conversion options to be acquired or generated. Today's computers use digital logic (see below and Chapter 2) and cannot work with analog signals directly. Detailed discussion of analog input and output issues is provided in Chapters 3 and 4.

1.1.2 Discrete Digital Signals

While analog signals can take on any value within a broad range (and so are called continuously varying), some parameters can occur in only two discrete states. Some of these include contact closure (yes/no), switch position (on/off), and many variations in which a complex phenomenon is characterized by a combination of many such status indicators (for example, the position of a rat in a maze as reflected by proximity switch positions).

Such parameters are easily represented electrically by assigning binary values (0 and 1) to two discrete voltage ranges (for example, less than 0.8 V and greater than 2.4 V). Both the meaning of the discrete positions and the quantitative voltage ranges are completely arbitrary. They serve merely as conventions to convey logical (Boolean) information.

Discrete digital inputs to a computer are generated by switches activated in an experimental set-up. Proximity switches or photodetectors can be used to convey physical position (here/not here) at any given point in time. Contact closures convey logical status about an experiment or test

(running/not running or valid/invalid) that can be used in conjunction with analog inputs to trigger and synchronize data acquisition. For example, discrete digital outputs to solenoids can be used to turn motors or other equipment on and off.

Since computers understand digital information, the only data-conversion problem with discrete digital signals is to ensure that logic level conventions are met at both the computer and device ends of the connection. Each parameter's status involves one signal and one "bit" of information. Computer interfaces with the capability to handle multiple such signals (typically 8, 16, or 32) at a time are frequently employed. The detailed issues involved in discrete digital I/O are covered in Chapter 5.

1.1.3 Time-Interval Measurements

Time-interval data represent a special need in laboratory interfacing applications. Certain information about real world phenomena can be represented in terms of magnitude or state using analog and discrete digital I/O modules. However, important information is often contained in the temporal pattern of magnitude or state changes, or in the interval between external events. In such cases, time intervals, in themselves, can be viewed as real world phenomena.

Realtime clock modules are used to measure or generate time intervals, just as other types of modules are used for input or output of analog or discrete digital data.

When operating in a measurement (input) mode, a realtime clock makes quantitative determinations of elapsed time between two external events. An input signal to the clock tells it when to start "ticking"; a second input tells it when to stop. The elapsed time (represented as the number of "ticks" that occured) between the two signal events can be obtained by the user's program from a register on the clock module.

When operating in an timebase generation (output) mode, a realtime clock produces the signal that triggers data acquisition at precise sampling intervals selected by the user. At the completion of each sampling interval, the clock transmits a signal to the analog or digital input/output device to trigger its next operation.

Since clock modules have on-board circuits to support digital inputs and to trigger other devices, they often contain Schmitt triggers, as well. These Schmitt triggers provide a mechanism for starting an I/O operation when a threshold voltage level is exceeded. In an analog data acquisition application, for example, you might wish to ignore data until

the input voltage signal exceeds 6.0 volts. A Schmitt trigger circuit on your realtime clock would allow you to select that threshold voltage level using a reference signal, and then trigger analog data acquisition only after the input voltage passed through 6.0 volts in the positive direction.

Realtime clock designs can also support other more sophisticated operations, such as "time stamping" input values with the time they were acquired, counting pulse train inputs, and measuring frequencies as well as elapsed time. These topics are covered in detail in Chapter 8.

1.2 Intelligent Instruments

Many modern laboratory instruments have built-in microprocessors which take over some of the realtime computing load in an application. For example, a chromatograph integrator may perform clocked A/D conversion on the detector output, characterize peaks in the acquired signal, and produce a printed report of the results. In most cases, the output of these intelligent instruments must be fed into a general-purpose computer for further analysis, report generation, or archival storage. Furthermore, many intelligent instruments require some degree of control (for example, parameter setting) which can be performed by a general-purpose computer. To move data and control information between a general-purpose computer and an intelligent laboratory instrument, some form of communication link between the computer and the instrument is needed.

The term *instrument interfacing* refers to techniques for controlling the communication link between a computer and an intelligent instrument. Though an intelligent instrument may solve many of the problems associated with handling raw realtime data (analog, discrete digital, and time-interval), interfacing the instrument to the computer may present a new and equally demanding set of problems. For this reason, this handbook includes descriptions of the most common types of communication links used for instrument interfacing.

1.2.1 Parallel Digital Interfacing

In contrast with discrete digital inputs, where each signal, or bit, represents a distinct and separate physical parameter, parallel digital signals are used to convey numerical values. The bit pattern simultaneously present on all lines of a parallel digital interface can be interpreted as a binary number or as a combination of decimal digits, each represented in binary form (Binary Coded Decimal). A series of data values can be transferred between a lab device and a computer using this format. The two devices employ some form of control signals or "handshaking" to ensure that each data value is correctly transferred.

The same hardware devices, called general purpose parallel digital interfaces, can be used to perform both discrete bit-level operations and true parallel-encoded data transfers. These devices and both kinds of applications are reviewed in Chapter 5.

1.2.2 IEEE-488 Interfacing

Various implementations of parallel digital interfaces provide different numbers of signal lines and a variety of handshaking protocols. These interfaces are compatible with each other only in the most general sense, and thus require considerable effort to establish actual data transfers. However, one type of parallel digital interface has been standardized by the IEEE for instrument interfacing and offers a much higher degree of base compatibility.

Defined in IEEE-488, this general purpose instrument bus (GPIB) defines an 8-bit wide data path, as well as 8 control lines, a communication protocol, and standards for mechanical designs and connector types. This makes it possible to connect devices with IEEE-488 interfaces and have them communicate successfully with minimal effort.

The IEEE-488 standard actually allows up to 15 separate devices to interface with, and thus share, the data and control lines. The communication protocol enables a master device to arbitrate among other devices requesting either service or permission to send or receive data. A computer can thus serve as controller of IEEE-488 operations and orchestrate the activities of up to 14 specialized lab instruments. Details of this type of operation are discussed in Chapter 6.

1.2.3 Serial Interfacing

Because parallel digital interfaces require many signal wires, they are relatively expensive to use for interfacing over distances of more than a few feet. An alternative is to use one signal wire for conveying data values encoded "serially" - that is, as a stream of single-bit values. Serial interfaces can convey information much less expensively over relatively long distances, but at much lower speeds than parallel interfaces, since they convey only one bit at a time.

There are two main categories of serial interface - synchronous and asynchronous. Synchronous interfaces use clocks at both ends of the connection to syncronize communications. This allows much higher speeds of data transfer at somewhat higher cost. Asynchronous interfaces use a handshaking or communication protocol to control data flow without clock syncronization. This allows widely disparate types of devices to communicate at a lower cost, but also at lower data rates.

Like parallel digital data, serially transmitted information can represent a stream of binary values. Often, though, it comprises a special alphanumeric coding called Serial ASCII (American Standard Code for Information Interchange), in which uppercase and lowercase letters, single-digit decimal characters, and commonly used control and punctuation characters are represented by a seven- or eight-bit code.

A standard has also evolved for logic levels, control signals, and other hardware aspects of serial asynchronous interfacing. It is embodied in the Electronic Industries Association's Recommended Standard 232 (EIA RS-232). RS-232 does not specify all aspects of interfacing; for example, connector type is left up to the implementor. Many vendors also implement a subset of the full specification. Thus, RS-232 interfaces tend to require more effort on the part of the user than IEEE-488.

Serial asynchronous interfaces are used for a wide variety of communication links in addition to lab devices. The same hardware and software standards support links from computers to display terminals, printers, plotters, and other computers. They are described further in Chapter 7.

1.3 How To Handle Realtime Data

Signal types and interfacing methods represent one dimension of the lab interfacing problem. Other equally important dimensions include:

- Connecting signal sources to modules
- Determining appropriate parameter settings
- Programming the I/O operation (and selecting the right programming tools)
- Managing and analyzing the incoming data
- Optimizing the application for performance

Chapter 2 of this handbook gives an overview of realtime modules, describing concepts and features that are common to all types of interfacing techniques. Chapters 3 - 8 present specific information on each interfacing technique. Information on how to connect the signal sources to the module, choosing appropriate parameter settings, and programming the module is presented. Chapter 9 addresses advanced concepts in realtime programming, including a section on buffer management techniques. Chapter 10 presents a discussion of factors affecting realtime performance. This information can help you in selecting the appropriate combination of hardware and software for your application, as well as in optimizing the performance of the selected system.

Only by addressing this combination of related issues can you create an overall application solution that best meets your needs. Even if you plan to purchase a turnkey package, understanding these issues can help you make the best selection. And, if you wish to build a flexible, general purpose research tool for the lab environment, the information in this handbook should help you solve most problems that arise.

Chapter 2
Realtime I/O Modules

In order to understand how realtime data is brought into a computer for subsequent analysis, you should consider the underlying architecture of computer systems. Computer systems are fundamentally modular in design, in that separate components of the system perform specific functions. The most basic components of a computer are the central processing unit (CPU), main memory, mass storage and I/O devices, and the system bus.

The *CPU* is the part of the system which actually performs arithmetic and logical operations on data values. The CPU also coordinates the operation of other parts of the system so that raw data and analysis results flow smoothly back and forth between the various system components.

Main memory is the repository for raw data and results that are being operated on by the system at any given time. Main memory also holds the instructions which make up the program being executed. The CPU fetches each instruction and any associated data values from main memory, performs the indicated arithmetic or logical operation on the data values, and returns the results to main memory. Each storage location in main memory is identified by a unique number, or *address*. Memory can be connected to the CPU via the system bus (see below), or via a high-speed, dedicated bus called a memory interconnect.

Mass storage devices, such as disks or tapes, are used for long-term storage of data and program instructions. Each mass storage device consists of a *controller* module and a physical storage medium. The CPU instructs the device controller to transfer data or program instructions between the physical storage medium and main memory. The controller then performs the transfer and notifies the CPU when the transfer is complete. The CPU passes instructions to the controller by way of *device registers*. Device registers appear to the CPU much like memory locations, in that they are storage locations identified by unique addresses. To communicate with a device controller, the CPU transfers information to or from the addresses representing the device's registers.

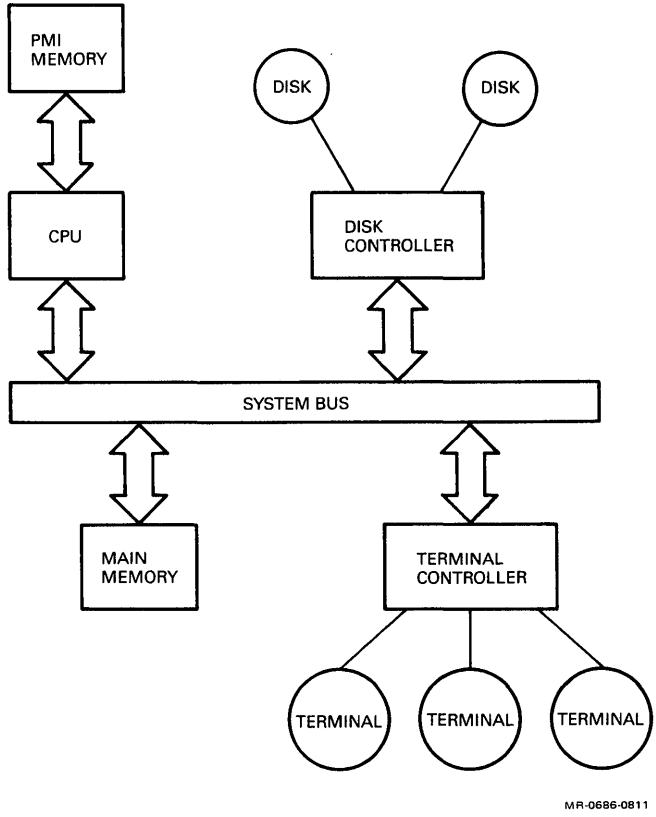
Input/output (I/O) devices include terminals, line printers, and other devices for moving information in and out of the system in alphanumeric or graphic form. Like mass storage devices, I/O devices are linked to the system through device controllers. Each I/O device controller has several registers which appear as addressable storage locations through which the CPU can control the operation of the device and transfer data. I/O device controllers typically have only one or two registers through which multiple data values are passed sequentially. Hence, they are sometimes referred to as data *ports*.

The *system bus* is the physical connection between the CPU, main memory, and device controllers by which communication between these parts of the system is accomplished. The bus consists of about 50 - 100 parallel digital signal lines. Subsets of bus lines are used to carry the following information:

- The address of the desired storage location or device register
- The value taken from or to be written to that storage location
- Various signals for control and synchronization of data transfers

The physical and functional relationships between the CPU, main memory, mass storage and I/O devices, and the system bus are illustrated in Figure 2-1.

Figure 2-1: Basic Computer Architecture



2.1 Realtime I/O Modules - What Are They?

For realtime data to be analyzed and/or stored by the computer, it must first be converted from its original form (analog, time interval, etc.) into a numeric format which is usable by the computer (usually binary). The data must then be moved into the system's main memory over the bus. Once in memory, data can be operated on by the program, moved to a

mass storage device, or both. On output, this process operates in reverse; data internal to the computer must be converted to an external format and delivered to an external device. Realtime option modules are used to perform the conversion between internal and external forms and, in some cases, to actually move the converted data into or out of memory.

Option modules for realtime I/O are similar to mass storage and standard I/O options in the following respects:

- They reside on the system bus.
- They are controlled by the program via device registers.
- Once activated, they perform their functions somewhat independently of the CPU.
- When an operation is completed, they can notify the program.

However, realtime option modules differ from mass storage and standard I/O devices in one important respect: instead of transferring data values between memory and a mass storage or standard I/O device, realtime modules transfer data to and from external instruments or devices, performing a conversion of some sort in the process. Thus, realtime option modules serve as bridges between the external world and the computer.

On the external side of this bridge, every realtime option module has a connector through which the physical link to the external device is made. This link consists of both data and control signal lines. In most cases, the connector is mounted directly on the module. The first step in implementing any application which uses a realtime option module is to develop a scheme for attaching the external signal lines to the appropriate pins of the on-board connector. This may be done using a multiconductor cable with the proper connectors at each end. If you foresee the necessity of repeatedly reconfiguring the external connections, you may want to lead the pins of the on-board connector to a break-out panel having screw, BNC, banana, pin-jack, or other quick connect/disconnect terminals.

The internal circuitry of a realtime option module performs several classes of functions.

- *Data translation.* This may involve a complex operation, like converting analog voltages to numeric representation; or it may involve simply converting external logic signal levels to levels which are compatible with the system bus.
- *Signal conditioning.* For example, some analog input modules have a variable gain amplifier to boost the level of the signal before converting it to numeric form.
- *Data buffering.* Some of the modules have fairly large (e.g., 1 Kbyte) first-in/first-out (FIFO) buffers to store data while waiting for the computer system or external device to become available.
- *Flow control.* Most modules have circuitry for synchronizing the flow of data to or from the external device (see Section 2.2).

The internal operation of realtime option modules is dependent on module type. For example, analog input modules might have internal amplifiers for on-board signal conditioning, but digital I/O modules would not. The internal features of realtime option modules vary from vendor to vendor and even across models produced by any given vendor. The types of features available on different types of modules are described in the chapters on each module type. You should look carefully at what features are available from various vendors and models when selecting a module for your application.

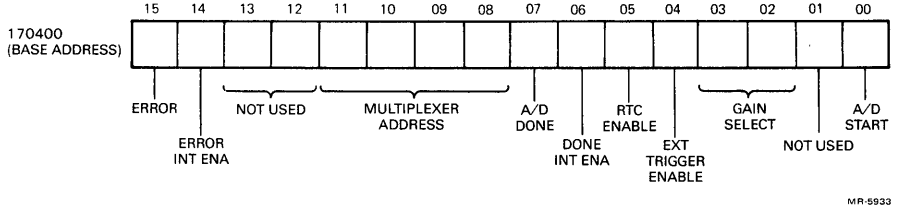
On the computer side of the bridge, a realtime option module has various registers through which control information and data are passed. As described above, these registers appear to the program as storage locations, each with a unique address. These device register addresses are all within a prescribed range of the physical address space of the system, called the *I/O page*. For example, on MicroVAX IIs the I/O page comprises the 2000 (hex) addresses beginning at address 20000000 (hex) in the physical address space of the processor. Device registers are almost always 16 bits wide. The addresses of the registers are even-numbered, though in some cases it is possible to access the low- and high-order bytes of a register individually.

Three kinds of registers are typically associated with realtime device modules:

- Control/status registers (CSRs)
- Buffer registers
- Address registers for direct-memory-access (DMA) operation

A *control/status register* is the main register through which the program controls a realtime option module. Individual bits and small groups of bits in the CSR are used to control such functions as channel selection or interrupt enabling. Some of the bits of the CSR may be "read-only" (not writable). When the program moves a new value into the CSR, the setting of any read-only bits remains unchanged, while the setting of writable bits changes according to the value written into the register. Other bits in the CSR may be "write-only" (always read as 0, regardless of their true value) or "read/write." Figure 2-2 shows the bit assignments of the CSR of an AXV11-C analog input module.

Figure 2-2: AXV11-C Control/Status Register



The kinds of functions which are typically programmable via the CSR are discussed in the chapters on each module type.

Buffer registers are used for transferring data back and forth between the program and the option module. For example, if an A/D module had been instructed, via the CSR, to perform an A/D conversion, the buffer register of the module would contain the data value when the conversion was complete. Similarly, to set the status of a block of 16 discrete digital output lines, a single 16-bit binary value might be written into a buffer register of a digital I/O module.

DMA address registers are used to pass the beginning main memory storage location and the number of data values to be transferred for a direct memory access transfer. DMA transfers are discussed in more detail in Section 2.2.2.

It is not uncommon for a realtime option module to have several sets of CSRs and buffer registers. This happens when the realtime option module comprises two or more independent devices. The AXV11-C module, for example, has a CSR and buffer register for analog input and two buffer registers for analog output. In these cases, reading or writing one set of registers has no direct effect on the status of any other set of registers on the module.

2.2 Flow Control

Synchronization of data flow between the external device and the computer is one of the important functions performed by any realtime option module. This function can be broken down into two somewhat independent operations: synchronization of data flow between the external device and the module, and synchronization of data flow between the module and the computer's memory. The basic mechanisms used for both types of synchronization are common among different types of realtime option modules. These basic mechanisms are described in this section. Additional details pertinent to specific module types are presented in Chapters 3 - 8.

2.2.1 Device/Module Synchronization

Triggering is the simplest form of flow control between an external device and a realtime option module. A trigger is a signal, usually a brief pulse, which indicates that data should be transferred. Trigger pulses may be generated at regular time intervals by some form of clock, or they may occur at irregular intervals determined by the occurrence of some external event, for example, the interruption of a photobeam. On input, when a trigger pulse occurs the realtime module performs a conversion, if necessary, and latches the data value in an on-board buffer. On output, data is converted, if necessary, and asserted on the module's output line(s).

Trigger events may also be generated by the computer system itself. (This is not to be confused with trigger signals generated by a clock module on the system bus, but acting somewhat independently of the program; such trigger signals would be classified as "external" in the present context.)

For example, the program might test the pH of a solution periodically and "trigger" data output to a solenoid-actuated valve if the pH value fell outside some prescribed range. In such cases, the occurrence of the trigger event might or might not be made known externally by the assertion of an additional output pulse. The important distinction, though, is that the computer system itself may generate trigger events in some applications.

A trigger, by its nature, is unidirectional and imperative. That is, when a trigger event occurs, it indicates to the realtime module that data must be transferred NOW. There is no mechanism for the realtime module to indicate that it is ready to transfer data or that data was successfully transferred following the trigger. Triggered data transfers are often used with analog input and output modules, and sometimes with discrete digital I/O.

Synchronization techniques which allow both the realtime module and the external device to signal readiness to transfer data and/or the successful completion of data transfer are called *handshaking* protocols. Handshaking protocols may be implemented using dedicated signal lines (hardware handshaking), or using the signal lines themselves (software handshaking).

Hardware handshaking involves the use of two dedicated signal lines to synchronize the flow of data. When the output device is ready to send a data value, it asserts a signal which signifies that a valid new data value is available for transfer. This signal is usually a brief pulse which is latched by the receiving device. When the receiving device has successfully captured the data value, it asserts a second signal which signifies that the data value has been received. The sending device waits for this return signal before asserting the next new data value. In this way, both devices participate in the flow control process. (Note that either the realtime option module or the external device may function as the "receiving" device in this scenario.)

This type of protocol is sometimes called *two-wire handshaking* since it involves the use of two dedicated signal lines. Some protocols involve the use of a third signal which signifies that the receiving device is ready to receive data. This is called *three-wire handshaking*. Figure 2-3 illustrates the flow of information under a three-wire handshaking protocol. A diagram of two-wire handshaking would be similar, but steps 1 and 2 would be omitted.

Figure 2-3: Three-Wire Handshaking Protocol

	INPUT DEVICE	OUTPUT DEVICE
1.	Asserts "ready to receive" signal (level)	
2.		Verifies "ready to receive" signal is asserted
3.		Asserts data on its output line(s)
4.		Asserts "new data ready" signal (pulse)
5.	Detects (latched) "new data ready" signal	
6.	Translates/latches data	
7.	Asserts "data received" signal (pulse)	
8.		Detects (latched) "data received" signal
9.		Loops back to step 2

Two- and three-wire handshaking protocols are used to synchronize data transfer on a word-by-word (or byte-by-byte) basis. This type of handshaking is often used by parallel digital communication modules.

A third type of hardware handshaking is sometimes used to synchronize sending and receiving devices at a coarser level. Basically, this involves the use of only the "ready to receive" signal line. The receiving device asserts this signal at a steady level whenever it is in a ready state. When it is unable to accept new data, for example if its internal buffer is nearly filled, it deasserts this signal to throttle the sending device. When the receiving device is once again able to accept input, it asserts the "ready to receive" signal to signify its readiness. The sending device checks the status of the "ready to receive" line before sending each data value. This type of handshaking is often used by serial data communication modules.

Software handshaking is similar to this third type of handshaking, but the signals used to throttle the flow of data are sent between devices over the data lines rather than over dedicated signal lines. For example, many serial devices use the ASCII characters DC1 (XON) and DC3 (XOFF) to signal "ready to receive" and "NOT ready to receive", respectively.

When the receiving device wants to suspend the flow of incoming data, it sends an XOFF character (hex 13; control-S) to the sending device. The sending device stops sending data until an XON character (hex 11; control-Q) is received, indicating that the receiving device is able to accept data once more. This kind of handshaking is often used for serial data communication.

2.2.2 Module/System Synchronization

The problem of synchronizing data transfers between a realtime option module and the main memory of the computer system is somewhat independent of device/module flow control. Once the data has been converted and latched into the module's buffer it must be moved to main memory. There are two general techniques for moving data from the option module to main memory: programmed I/O (PIO) and direct memory access (DMA).

Programmed I/O refers to situations in which the program controls the transfer of each data value from the module to memory through the execution of processor instructions. On input, whenever a new data value is present in the buffer register, the module notifies the program. The program then transfers the data value from the buffer register to a storage location in the computer's main memory.

Notification that a new data value is ready can be passed to the program in two ways:

1. When the conversion is complete, one of the bits of the CSR, the "done" bit, is set. The program must repeatedly test the CSR to determine whether the done bit is set. When the setting of the done bit is detected, the program transfers the data value from the module's buffer register to main memory. To maximize realtime response, a large percentage of CPU time must be devoted to testing the done bit. This is called *polled I/O*.

2. When the conversion is complete, the module generates a hardware interrupt of the system. This causes program control to be transferred to an "interrupt service routine," which moves the data value from the module's buffer register to main memory. While the CPU is waiting for the interrupt to occur, it can be performing other tasks, such as reduction or transfer to mass storage of previously acquired data. This mode of operation maximizes the use of the computer's resources, but may be relatively slow due to the overhead associated

with switching control to the interrupt service routine. This is called *interrupt-driven I/O.*

Direct memory access refers to situations in which the option module transfers data directly between the computer's main memory and the external device. The program loads the DMA address registers of the module with the physical memory location at which data storage is to begin, and loads the number of data values to be transferred into another device register. The CSR of the module is then loaded with information specifying various other parameters for data acquisition, and the acquisition process begins. As the data are collected, data values are transferred directly from the option module to the designated main memory storage locations. During this period, the CPU can devote its attention to other tasks. When all data have been thus transferred, the module notifies the program.

Both the DMA device and the CPU use the system bus to get information into or out of main memory. Since both need time to "think" between memory transfers, sharing the bus is usually not a problem. While the CPU is performing arithmetic or logical operations on data, the DMA device can use the bus to transfer data to or from memory. Similarly, while the DMA device is loading data from the external device into its internal memory, the CPU can use the bus for memory transfers or device communication. When both want to use the bus at the same time, a bus arbitration scheme is used so that neither the CPU nor the DMA device takes over the bus for an extended period of time to the exclusion of the other. Since the speed at which the bus operates is typically much faster than the speed at which either the CPU or the DMA device runs, sharing the bus usually has little impact on overall system performance. However, if several DMA devices were running simultaneously, the speed of the bus might become a limiting factor.

DMA operations are highly efficient in that much of the overhead of the data acquisition process is off-loaded from the CPU. Typically, very high data acquisition rates are possible. However, modules which support a DMA mode of operation are significantly more expensive (by a factor of 2 to 3) than modules which support only PIO modes. Furthermore, special programming techniques must be used for efficient DMA operation, particularly when successive blocks of data are to be acquired continuously.

2.3 Hardware Configuration Of Modules

As described above, a realtime option module has various registers which, to the program, look like storage locations. Each register is assigned an address which uniquely identifies that particular register on a specific module. If a module has several registers, their addresses are usually contiguous. For example, the addresses of various registers on an AXV11-C module might be as follows:

Address (octal)	Register
176400	A/D Control/Status register
176402	A/D Buffer register
176404	D/A Buffer register A
176406	D/A Buffer register B

Since a register's address is like a "name" to which it responds, the address assigned to each register is an intrinsic characteristic of the module, rather than being determined by the CPU or some other part of the system. Thus, when you unpack a new option module, the addresses of the registers are built into the hardware or firmware of the board.

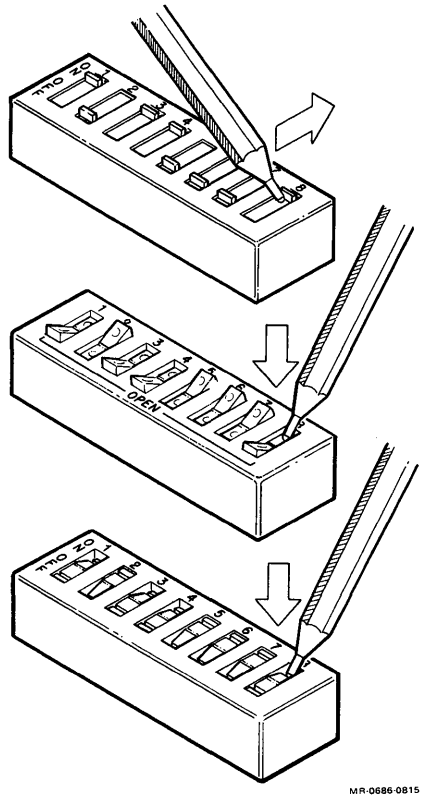
But what if you wanted to put more than one of the same type of module in your system? Wouldn't there be some confusion as to which module was being addressed when a program sent a data value to storage location 176404, for example? There would be, unless you were to first change the addresses of the registers on one of the modules to eliminate the conflict.

Virtually all option modules allow you to change the addresses of module registers to nonstandard values. For modules with more than one register, changing the address of the first register causes all other register addresses to change in parallel. Register addresses are changed by installing or removing certain connections on the board, called *jumpers*, which are designed for easy modification. Jumpers are connectors or wires running between binding posts or wire-wrap terminals. On some boards, they must be soldered in or cut out using fine wire-cutters. On other boards, they can be installed or removed using wire-wrap tools or needle-nosed pliers. The location of the jumpers and the scheme which equates jumper configuration to register addresses is documented in the user's manual for any particular module.

In addition to register addresses, many realtime modules have other features which can be selected or modified by changing jumpers on the board. Which features are jumper-configurable depends on the type and model of module. For example, some A/D modules have on-board amplifiers which boost the incoming signal before it is digitized. The gain of such amplifiers is often determined by the configuration of jumpers.

In cases where a feature might be changed frequently, some manufacturers use banks of tiny switches, called DIP-switches, instead of jumpers to facilitate reconfiguration. The appearance of DIP-switch packs varies from one manufacturer to another. Figure 2-4 shows how to set three representative types of DIP-switches.

Figure 2-4: DIP Switch Types



When selecting a realtime module, it is advisable to determine which features are jumper configurable, which are programmable, and which

are not modifiable. Modules which have configurable or programmable features are more adaptable to changing application needs.

2.4 Programming Realtime Option Modules

Up to this point, the description of realtime option modules has focused on the internal workings of modules and their functional relationship with other parts of the computer system. With this knowledge in hand (and additional information about the operational details of specific module types to be covered in subsequent chapters), the user must address the question of how to write programs for controlling realtime option modules. There are several approaches to module programming which may be viewed as heirarchically related:

1. Direct program manipulation of device registers
2. System service calls to standard device drivers
3. High-level language subroutine calls
4. User-friendly interfaces

The succeeding sections contain an overview of these different approaches to option module programming.

2.4.1 Direct Device Control Routines

The most basic level at which modules can be programmed is by direct manipulation of device registers. To accomplish device control at this level, the programmer must have a detailed understanding of the architecture and functionality of the module. In addition, a fairly sophisticated understanding of overall system architecture and programming is required, particularly for multitasking and memory-mapped systems such as MicroVAX/MicroVMS. Direct device control routines are typically written in assembly language, though FORTRAN or BASIC "peek" and "poke" utilities can be used for loading and reading device registers.

When programming at the direct control level, the user controls the operation of the module by setting or clearing bits in the CSR, and loading addresses or data into the other device registers. The sequence and timing of device operations can be tightly controlled, within the hardware limits of the module. The user has maximum opportunity to

interleave other processing steps with discrete module I/O steps. Thus, programming at this level gives the user the greatest degree of flexibility and, often, speed.

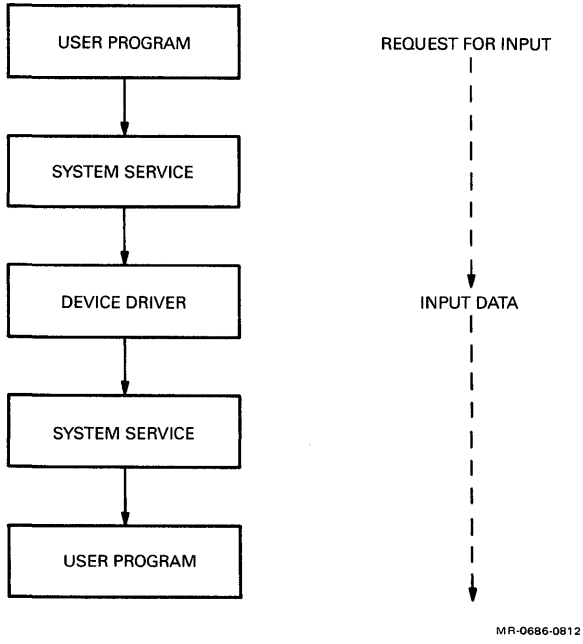
2.4.2 Device Drivers

Every operating system has services for performing device I/O operations. These services can be viewed as an intermediate software level between a user's program and blocks of code for controlling devices. For example, a user program might contain the FORTRAN statement:

```
READ(5,100) A, B, C
```

At run time, the request for input is passed to a block of system service code. The system service in turn passes control to a block of code which actually performs the console terminal I/O operation by manipulating the hardware registers of the console device (logical unit 5 by default). This bottom level of code is called a *device driver*. The device driver returns data values and status information to the system service code which then passes the data back to the user's program. This process is illustrated in Figure 2-5.

Figure 2-5: Device Driver I/O



This scheme has several advantages. First, the system service can process I/O requests to or from many devices. If the user program had previously assigned logical unit number 5 to a disk file, the system service would use the device driver for the disk rather than the console device driver (transparently to the user).

Second, the system service can coordinate the I/O request with other things that might be happening in the system. For example, if another process were using the same disk at the time the request was received, the system service might place the current request in a queue for later processing. Finally, the system service can perform various error checks and take appropriate action if an error condition occurs.

Just as terminals or mass storage devices can be controlled using this system service/device driver mechanism, realtime I/O modules can be controlled in a similar fashion. In order to accomplish this, though, the user must have a device driver for the given I/O option module. Such a driver would have to conform to all the conventions used by the operating system service for argument passing, asynchronous operation, and error handling. Since these conventions are often complex, the writing of device drivers can be a tedious programming task. Furthermore, there may be a significant amount of overhead associated with the system service. That overhead could be damaging to the requirement for predictable and fast response to realtime events. In exchange for these difficulties, the system service/device driver mechanism offers enhanced safety and convenience.

While the job of writing a device driver may be beyond the capability of many application programmers, drivers are often supplied by the realtime module vendor. In that case, the user need only determine whether the realtime performance limitation imposed by the system service/device driver is too severe for a given application. Performance information under a variety of conditions is typically supplied by the vendor to aid the user in making this determination.

2.4.3 Subroutine Libraries

A third programming technique for controlling realtime I/O devices is to use special-purpose subroutines for performing common, basic realtime I/O operations. Device control within realtime subroutines can be based either on direct device control routines or drivers. Such subroutines can be used as program modules, linked in with the user's main program and other user-written or canned subroutines.

Vendors of computer systems or realtime option modules usually supply their customers with a collection of subroutines for performing common realtime I/O tasks. Ideally, such packaged subroutines are callable from several programming languages. User-written direct device control routines are often in the form of a subroutine.

A subroutine call is relatively easy-to-use, since the end-user need understand only the principles and not the details of module operation. Furthermore, realtime subroutines usually offer high-level control, combining many basic operations in a single call. For example, the FORTRAN call

```
CALL FASTAD( ICHAN, IRATE, ICOUNT, IBUFFER, NPTS)
```

might perform clocked A/D conversions, specifying the channel number (ICHAN), the clock rate (IRATE, ICOUNT), the buffer address into which the data values were to be written (IBUFFER), and the number of data points to convert (NPTS). (A full discussion of these concepts is given in Chapter 3.)

When using vendor-supplied subroutines for performing realtime I/O operations, the user may find that the performance or functionality offered in a subroutine package does not match the needs of the application. The "match" depends on the demands of the application and the cleverness of the programmer who wrote the subroutines, assuming that the desired performance/functionality is within the capacity of the computer system and the option module. If the demands of the application are not met using available subroutine packages, the user may find it necessary to write a specialized direct device control routine.

2.4.4 User-Friendly Interfaces

The highest level of realtime device control is through menu- or icon-driven "programless" interfaces. Such interfaces are standalone programs which guide the user through the selection of various realtime parameter values (for example, sampling rate and number of channels) and also allow the specification of postacquisition data handling (graphic display, analysis, and/or storage).

As with subroutine packages, performance and flexibility are traded off against ease-of-use. Applications which are basically linear in nature (collect the data, analyze the data, store the data/results) are often handled very effectively by user-friendly interfaces. Applications which demand even simple interconnections between multiple realtime events may be impossible to implement with a canned interface (for example, "...when a threshold temperature is reached, turn off the heater..."). Here again, the user may find it necessary to fall back to a lower level of control in order to perform the application.

2.5 Intelligent Instruments

Realtime option modules, as described in the preceding sections, are used to handle raw experimental data from laboratory devices: analog, discrete digital, and time-interval data. Built-in microprocessors, which perform part of the data acquisition/reduction task, are part of many intelligent instruments. An intelligent device may be wholly self-sufficient. Once started, it may regulate various operating parameters, acquire the raw data, analyze the data, and display the results on a CRT or printer. Many modern chromatographs and spectrophotometers have this degree of functionality.

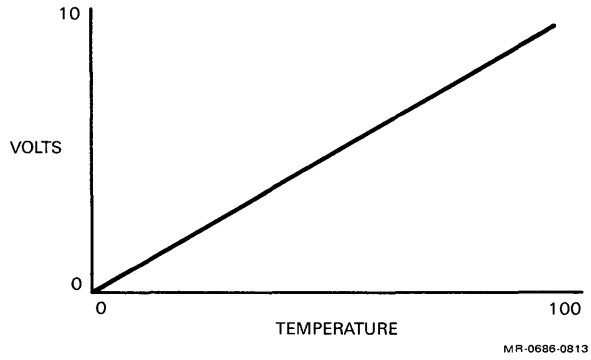
When an intelligent instrument is used to perform some part of an experiment or analytic procedure, it may be desirable to feed its output directly into a host computer for additional analysis or data management. Furthermore, an intelligent instrument may require some degree of external control - to initialize or regulate operational parameters, for example. In either case, the instrument must be connected to the host computer by some form of communication link. Commonly used communication links are serial (EIA), parallel digital (BCD, binary), and IEEE-488 (General Purpose Instrument Bus).

In comparison with realtime option modules, intelligent instruments are generally easier to interface to a computer, since many of the time-critical operations can be done by the instrument. Furthermore, the communication links most commonly used are reasonably well standardized with respect to signal levels and protocols. Thus, intelligent instrument interfacing, regardless of the type of communication link used, is fundamentally different from interfacing using realtime option modules. However, the serial or parallel signal, like a raw analog or discrete digital signal, must be brought into the computer through an option module of some kind. Despite standardization, the connections and protocols can be frustratingly difficult to follow. For these reasons, chapters covering the most common forms of communication links to intelligent instruments have been included in this handbook.

Chapter 3
How To Perform Analog Input

It is common practice in the modern laboratory to represent various physical, chemical, or physiological properties of a system under study as electrical signals. Usually, the signal is a voltage level which varies continuously over some range as the real-world quantity that it represents changes. For example, the diagram below shows the relationship between temperature and the signal produced by an ideal temperature transducer.

Figure 3-1: Output of an Ideal Temperature Transducer

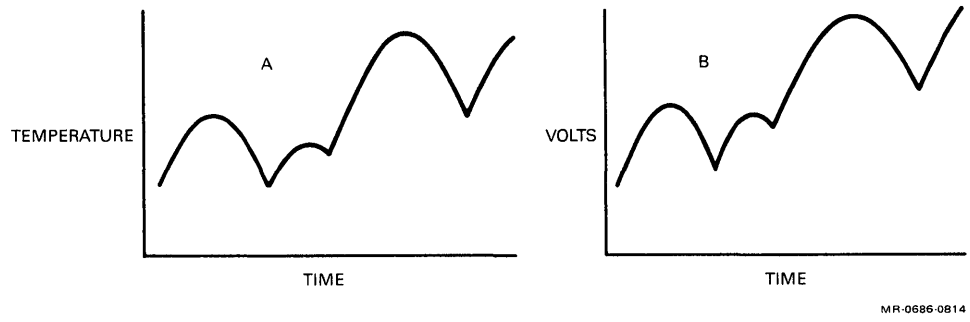


The range of the output is 0 to 10 volts. Within that range, each possible voltage level corresponds uniquely to some precise value of the quantity it represents (that is, temperature). In this idealized case, a signal level of 1 volt represents 10 degrees centigrade; 2 volts represents 20 degrees, and so on. Since the temperature can be any value between 0 and 100 degrees, the signal can take on any value between 0 and 10 volts.

If the temperature followed some varying pattern over a period of time, for example, an hour, a plot of temperature against time might look something like Figure 3-2A. A plot of the corresponding signal produced by the transducer is shown in Figure 3-2B.

The two shapes are identical. An observer might measure the voltage. Since the pattern of voltage changes accurately represents temperature variations in the real world, the voltage readings can be converted to temperatures, provided of course that the algebraic relationship between temperature and voltage is known. The signal, therefore, provides a model, or analog, of some real physical property. Hence, signals of this type are called *analog signals*.

Figure 3-2: A Time-Varying Analog Signal



3.1 Digital Representation Of Analog Signals

Practically speaking (quantum theory aside), analog signals and the real-world phenomena they represent are continuous in both amplitude and time, within experimental bounds. That is, an analog voltage can assume any of an infinite number of possible levels, and can be measured at any of an infinite number of points in time. Digital computers are only capable of processing data in discrete form, though. For this reason, analog-to-digital conversion necessarily involves transformation of the continuous analog signal to a numeric form which has discrete, finite resolution in both the amplitude and time dimensions. In the following sections, the transformation of analog signals to discrete numerical values will be discussed, first with regard to amplitude, and then with regard to time.

3.1.1 Digital Representation Of Signal Amplitude

For analog signals to be processed by a computer, voltage levels must first be converted to numbers in a form which the computer can process. Though numbers can be represented in many ways in digital computers, A/D converters almost universally convert voltages to integer values in binary form. For example, an A/D converter module might convert voltages in the range 0 - 10 V to integer values in the range 0 - 4095. In order to convert those integer values to real-world equivalents, two algebraic relationships must be considered: the relationship between the real-world parameter value and voltage, and the relationship between voltage and the intermediate numeric values produced by the A/D converter. In this section, the second of these two relationships will be discussed - that is, the factors which determine how analog voltages are converted to numeric values.

On the input side, any A/D converter accepts only voltages which fall within a prescribed range. Voltages which fall outside the input range of the module will be improperly converted and may damage the module. Common input ranges of A/D modules (in volts) are 0 to +20, 0 to +10, 0 to +5, -5 to +5, and -10 to +10. These ranges can be characterized by their absolute magnitude (5, 10, or 20 volts) and by their polarity (unipolar or bipolar). Many A/D modules allow you to select both the absolute range and the polarity of input voltages using jumpers.

On the output side, the converted values also have a restricted range. The numeric range of the output is determined by the number of binary digits, or bits, in the converted values. The most common A/D converters encountered in laboratory applications convert voltages to 12-bit binary numbers. Since the largest positive number that can be expressed as a 12-bit binary number is 4095 (decimal), the *numeric precision* of a 12-bit A/D converter is said to be 1 part in 4096 or 1:4096 (remember that the number 0 is valid, too). Both 14- and 16- bit A/D converters are also available for UNIBUS and QBUS systems, having numeric precisions of 1:16384 and 1:65536, respectively.

Even if you select an A/D module with the highest available numeric precision, the resolution of the converted values will still be finite. The implication of this finite resolution is that each discrete digital value corresponds to a range of analog signal levels. Taking the case of a 12-bit A/D converter with an input voltage range of 0 - 10 volts, Table 3-1 shows the relationship between analog signal levels and nominal numeric representation.

Table 3-1: Numeric Representation (example)

All voltages greater than:	but less than or equal to:	are represented by the number:
- ∞	0.00122	0
0.00122	0.00366	1
0.00366	0.00610	2
0.00610	0.00854	3
.	.	.
.	.	.
.	.	.
9.99146	9.99390	4093
9.99390	9.99634	4094
9.99634	∞	4095

In this example, the smallest voltage difference which can be resolved after conversion is 0.00244 volts or 2.44 millivolts. This is called the *voltage resolution* of the A/D converter. The voltage resolution of an A/D converter is determined by two parameters: the numeric precision of the converted values and the input voltage range.

The higher the voltage resolution of the A/D converter, the more faithfully small variations in the amplitude of the analog signal are represented. Table 3-2 gives the voltage resolution for various combinations of absolute input range and numeric precision.

Table 3-2: Voltage Resolution

If the absolute input range is:	and the numeric resolution of the A/D is:	the voltage resolution of the output will be:
5 volts	12-bit - 1: 4096	1.22 millivolts
"	14-bit - 1:16384	305. microvolts
"	16-bit - 1:65536	76. microvolts
10 volts	12-bit - 1: 4096	2.44 millivolts
"	14-bit - 1:16384	610. microvolts
"	16-bit - 1:65536	153. microvolts
20 volts	12-bit - 1: 4096	4.88 millivolts
"	14-bit - 1:16384	1.22 millivolts
"	16-bit - 1:65536	306. microvolts

A final variation on numeric representation of analog voltages applies to modules configured for bipolar input. Some modules allow you to select either offset binary or two's complement notation for the digital output.

With *offset binary* notation, all voltages, whether positive or negative, are represented as positive integer values. A negative full-scale voltage is nominally represented by the number 0. A positive full-scale voltage is nominally represented by the number (2**n)-1 , where n is the number of bits in the converted value. A voltage of 0 is nominally represented by the number (2**(n-1))-1. Thus, the A/D converter behaves as though an offset equal to the positive full-scale voltage has been added to the input before conversion.

With *two's complement* notation, positive voltages are represented by positive integers; negative voltages are represented by negative (two's complement) integers, and 0 volts is represented by the number 0. Binary offset and two's complement notation for a 12-bit A/D converter with an input range of +/- 10V is shown in Table 3-3.

Table 3-3: Binary Offset and Two's Complement Notation

Input voltage	Binary offset notation:		Two's complement notation:	
	octal	decimal	octal	decimal
10 V	007777	4095	003777	2047
0 V	003777	2047	000000	0
-10 V	000000	0	174000	-2048

In either case, an algebraic transform would ordinarily be applied to the numeric output to convert to physical units (for example, degrees centigrade). Since the transform can account for the offset (or lack of one), the choice between offset binary and two's complement output notation is somewhat arbitrary. Once you have established a convention, though, you should select A/D modules which follow that convention, or which are jumper-configurable.

3.1.2 Digital Representation Of The Time Dimension

Often, the relevant information about the real-world phenomenon under study is contained in the pattern of variation of an analog signal over time, rather than in a single-point value. For example, if a thermocouple is implanted in the nasal cavity of an experimental animal, the pattern of change in the the amplitude of the signal over time reflects the animal's breathing pattern, as successive inhalations and exhalations cool and warm the probe. In order to extract information about breathing, as opposed to the instantaneous temperature of the airstream, it is necessary to have a record of the thermocouple output voltage over time.

Like the amplitude dimension, the time dimension of an analog signal is continuous and infinitely divisible. That is, within any given interval, the amplitude of the signal can be measured at an infinite number of points in time. Since computers can process only discrete data, the continuous time dimension of the analog signal must be translated into discrete form. The time dimension of an analog signal can be represented in two ways:

implicitly by position in a data array, or explicitly by timestamping each data value.

With an *implicit time dimension*, the amplitude of the signal is determined at known, usually equally-spaced, points in time. To accomplish this, successive A/D conversions are triggered by some form of clock, often a separate module on the system bus alongside the A/D module. The resulting values are placed in successive memory locations, either under program control as each conversion is triggered, or in blocks of values by direct memory access (DMA). Subsequently, the discrete time-point associated with each value can be determined by reference to the ordinal position of the value in the memory array.

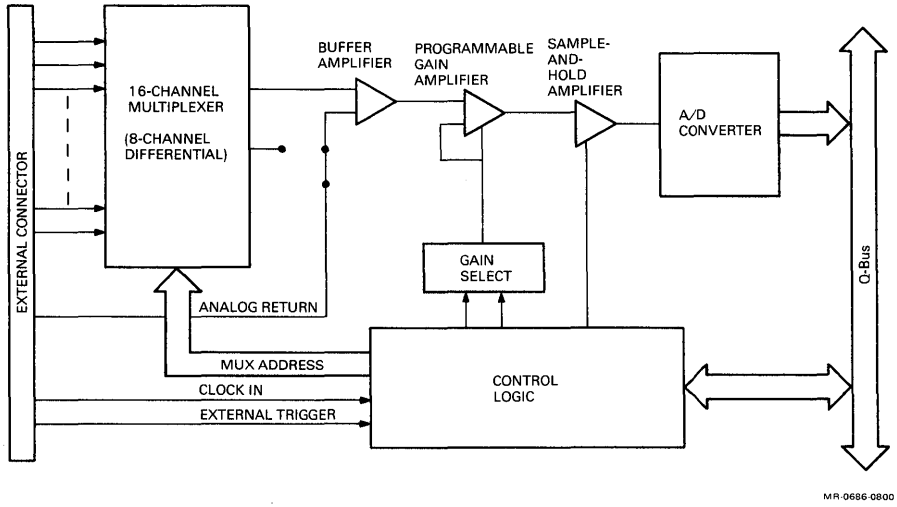
With *timestamping*, both the numeric amplitude of the signal and a clock reading is stored for every sampled point. This is ordinarily done under program control. The data values and clock readings can be stored in successive pairs of memory locations or in two separate arrays. Timestamping is particularly useful in situations where conversions are triggered at irregular intervals by some external event such as a threshold crossing of a second signal. However, timestamping entails greater overhead, in terms of both storage requirements and data acquisition speed. For this reason, you should use timestamping only when an implicit time dimension does not meet the application requirements.

3.2 Analog Input Modules

All analog input modules have a control/status register (CSR) and a data buffer register (DBR) which are addressable as word-length storage locations in the system I/O page. The DBR and some of the bits in the CSR are read-only. The CSR is used to control the operation of the module and to pass status information to the program. The DBR is used to pass digitized data values to the program.

Figure 3-3 shows a simplified block diagram of the ADV11-C. This module is typical of non-DMA analog input modules; indeed, the internal operation of this module is identical to the analog input side of the AXV11-C.

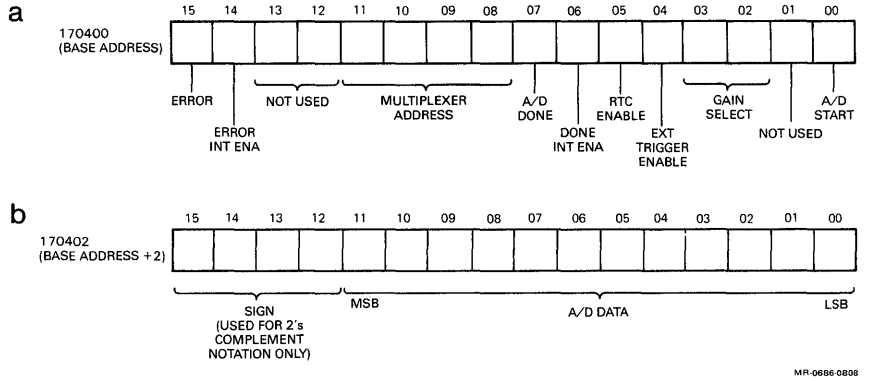
Figure 3-3: Simplified Block Diagram of the ADV11-C Analog Input Module



The incoming analog signal lines are led first to a multiplexer (MUX) circuit. One of the 16 signal lines is passed through to the buffer amplifier. The program selects the MUX channel by setting a bit-field in the CSR. Jumpers on the module are used to determine whether the module is configured for single-ended or differential input (see Section 3.3.1). The output of the buffer amplifier, carrying the analog signal from the selected channel, is led to a programmable gain amplifier. The program selects the gain factor by setting a bit-field in the CSR.

A conversion is triggered by an input from the clock or external trigger, or by the setting of a bit in the CSR under program control. When a conversion is triggered, the control logic signals the sample-and-hold amplifier (SHA) to clamp the amplified signal. The function of the SHA is to hold the signal going to the A/D converter at the value which is asserted at the moment the control logic issues the command to perform a conversion. When the conversion is complete, the control logic sets a bit in the CSR and, optionally, requests a Q-bus interrupt. Figure 3-4A shows the bit assignments in the CSR.

Figure 3-4: ADV11-C Control/Status and Data Buffer Registers



By writing or reading bits in the CSR, the program is able to control the functioning of the module and obtain certain status information as follows:

1. Select the trigger source (bits 4-5) or start a conversion immediately (bit 0).
2. Select the gain factor (bits 2-3); the ADV11-C and AXV11-C are capable of gains of 1X, 2X, 4X, or 8X.
3. Select the channel on which conversion is to be performed (bits 8-11).
4. Detect the completion of the conversion (bit 7).
5. Detect the occurrence of an error in conversion - for example, trying to start an A/D conversion during multiplexer settling time (bit 15).
6. Enable either or both of the above two status conditions (items 4 and 5) to cause a Q-bus interrupt to be generated (bits 6 and 14).

The converted data value is returned in the DBR. The format of the data is shown in Figure 3-4B. Note that, though the converted data value is only 12 bits in length (for these particular modules), a full word - 16 bits - is returned to the program. The high-order bits are used for sign extension if the module is configured for two's complement notation; that is, the value of bit 11 is replicated in bits 12-15.

So far, we have discussed the properties of analog signals and given an overview of the process of analog-to-digital conversion. What remains, of course, is all the practical details of how to capture an analog signal for the purpose of extracting the desired information. Briefly, the steps involved in this process are as follows:

- Connecting the input signal to the A/D module
- Controlling noise in the analog signal
- Selecting a gain factor
- Selecting a sampling rate
- Selecting a triggering mode
- Writing a program which sets the relevant parameters, initiates data acquisition, and stores the data for subsequent analysis

At every step in this process, the information content of the signal must be preserved. As described above, information may be contained in both the amplitude and time dimensions of the signal. The remainder of this chapter deals with the various issues you must address to preserve the information content of the signal.

3.3 How To Connect The Signal Source To The Option Module

Devices which produce analog signals generally fall into one of two categories: *grounded-source* devices or *floating-source* devices. The way you connect the signal to an A/D module depends on whether the source is grounded or floating. A floating-source device has no electrical path to ground; a grounded-source device has an electrical path to ground. The ground may or may not be earth ground. That is, a device might be grounded to a chassis or other structure which is not earth-grounded.

Some devices are grounded through their power cords; others have a separate grounding wire connected to a known ground. Note that the distinction between grounded- and floating-source devices is based on whether the circuit that generates the signal is grounded, not the chassis. Some devices have separate leads for grounding the chassis and the circuit.

An analog voltage is always measured as a voltage difference between two nodes in a circuit. Thus, when an A/D module is used to digitize a voltage signal, both nodes of the signal source must be connected so that a circuit through the A/D module is completed. This can be accomplished using several input connection schemes:

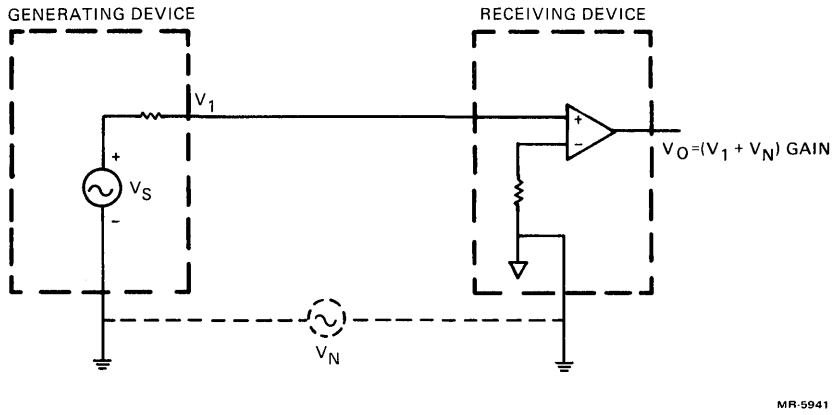
- Single-ended inputs
- Pseudo-differential inputs
- Differential inputs

3.3.1 Single-Ended Connection

Single-ended analog inputs have one side of the analog source connected to the A/D converter amplifier and the other side connected to ground. A single-ended input scheme is used when the signal source is grounded. The "active" node is connected to one of the inputs of the A/D module, and the signal ground of the device is connected to the signal ground of the module to complete the circuit. If several signals from the same device are being measured, only one lead for each signal is needed plus a single lead for connecting the circuit grounds of the device and the A/D module. Hence, this is called single-ended input. Figure 3-5 illustrates single-ended input.

The benefit of a single-ended input scheme is that you get twice as many channels as with a differential input scheme. The disadvantage is the loss of the common mode rejection that is available with a differential scheme (see Section 3.4.1). Therefore, a single-ended input scheme should be used only when the following conditions are met:

Figure 3-5: Single-ended Input Diagram



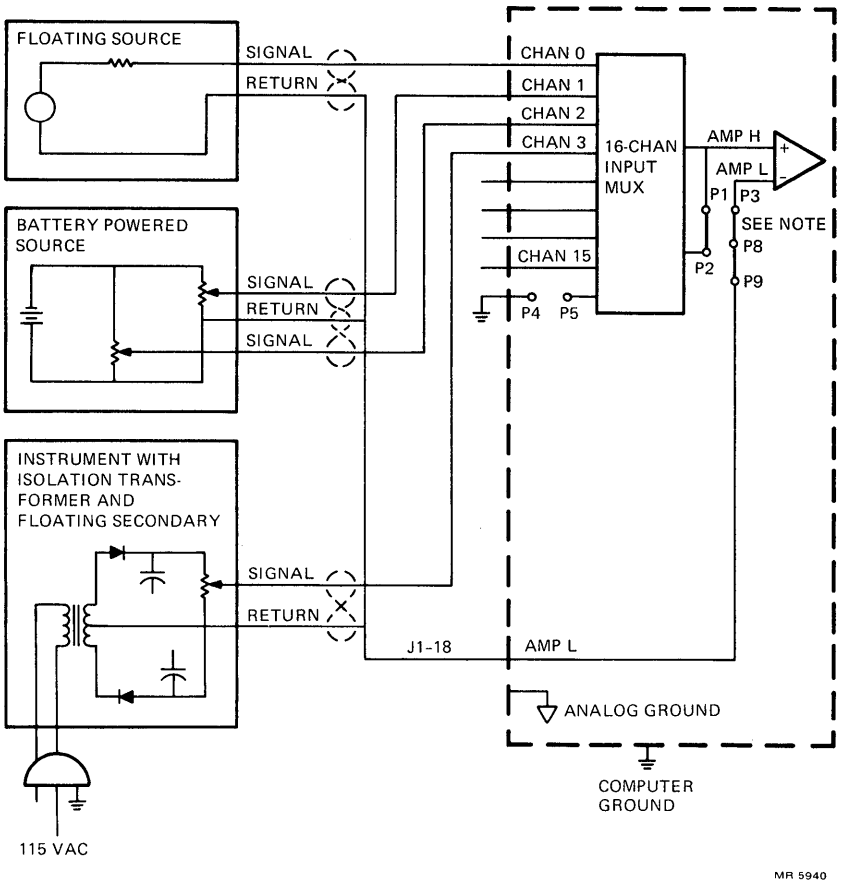
- The input level is higher than 1 V.
- Input cable lengths are less than 15 feet long.

The signal source may be located some distance from the computer, and a voltage difference may occur between the source ground and the computer ground. This ground voltage difference is included in the signal received by the A/D converter. To decrease this ground difference, plug the device into an AC receptacle on the same power circuit and as close as possible to the receptacle providing power to the computer.

3.3.2 Pseudo-Differential Connection

Pseudo-differential analog input is a variation of single-ended input. A pseudo-differential input scheme can be used with floating-source signals. To implement this input scheme, configure the A/D module jumpers for single-ended input. Then, connect the return sides of all inputs to the low side of the A/D converter's input. Finally, connect the common return externally to analog ground through a 1 - 10 kOhm resistor. Figure 3-6 illustrates pseudo-differential input.

Figure 3-6: Pseudo-Differential Input Diagram



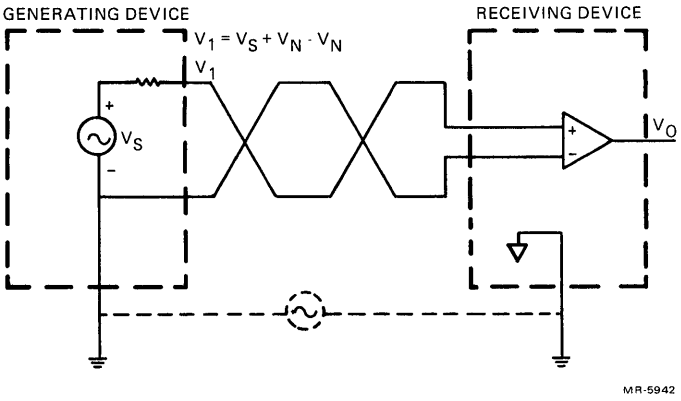
Pseudo-differential input provides some degree of common-mode noise rejection (see Section 3.4.1), without sacrificing the number of input channels required for differential input. A pseudo-differential input scheme should be used when the following conditions are met:

- The input level is higher than 100 mV.
- Input cable lengths are less than 25 feet long.

3.3.3 Differential Connection

Differential analog inputs have one side of the signal source connected to the positive input of an A/D channel, and the other side connected to the negative input of the same A/D channel. A differential input scheme is used when the signal source is floating. To complete a circuit, both the positive and negative sides of the signal source must be connected to ungrounded inputs of the A/D circuit. The voltage which is digitized is actually the difference between the voltages at each of the nodes (that is, each with reference to the source's circuit ground). Hence, this is called differential analog input. Figure 3-7 illustrates differential input.

Figure 3-7: Differential Input Diagram



The voltage at the negative input is subtracted, in analog fashion, from the voltage at the positive input before digitization. When the A/D converter is configured for a bipolar input range, the order in which the source leads are connected to the A/D inputs affects only the apparent polarity of the signal. However, when the A/D module is configured for a unipolar

input range, improper ordering of the source leads causes the signal to appear out-of-range (usually 0 volts) at all times.

The benefit of a differential input scheme is that noise voltages appearing at the same time on both sides of the source are rejected by the A/D input amplifier. The disadvantage is that the number of input channels is reduced to half of what would be available for single-ended or pseudo-differential inputs. A differential input scheme should be used when the following conditions are met:

- The input level is lower than 100 mV.
- Input cable lengths are more than 25 feet long.
- Low impedance, twisted-pair, shielded cables are used.

3.4 How To Control Noise In Analog Transmission

Noise is any unwanted component present in the data signal. Digital transmissions generally are less prone to noise problems than analog transmissions. Digital transmission levels are relatively high, and impedances are low. Analog transmissions, however, often require high precision and may occur at low voltage levels. Moreover, source impedances on external analog equipment are often relatively high, and associated circuits display a correspondingly increased susceptibility to noise pickup.

Noise usually manifests itself as a distortion of the smooth continuity of the data signal. For example, a data-producing device might display a signal such as that shown in Figure 3-8A. By the time the signal reaches the A/D module, however, it may resemble the signal shown in Figure 3-8B. It is clear that much of the information in the original signal has been lost.

In low-level voltage situations, noise can be an important problem, since the noise voltage level may be a significant percentage of the data voltage level. In addition, if the signal is being amplified for greater resolution, the noise is amplified also. For example, a signal such as that shown in Figure 3-9A might look like the one in Figure 3-9B after it is amplified.

Figure 3-8: Analog Signal with Noise Component

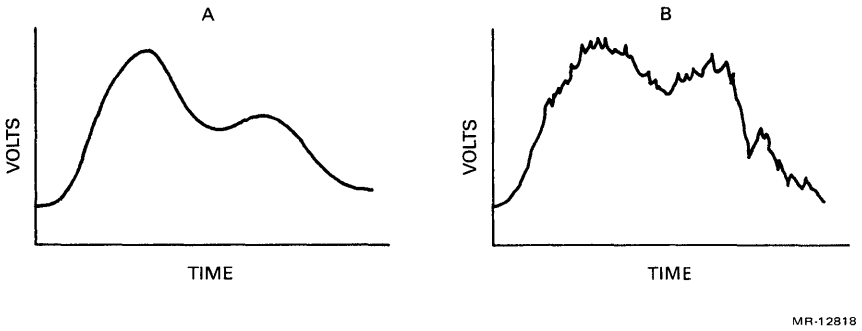
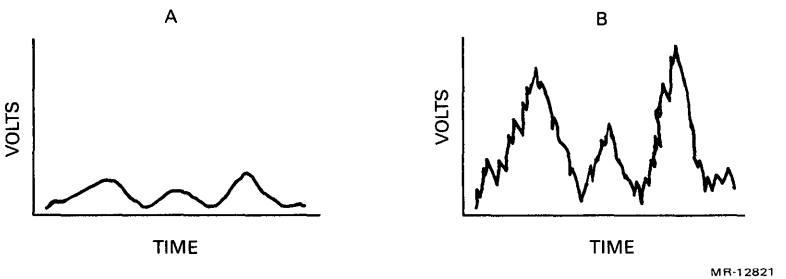


Figure 3-9: Amplified Signal with Noise Component



The amount of noise introduced into the signal is often related to the distance the signal travels before it reaches the A/D module. A great deal of noise can be eliminated by keeping cable lengths to a minimum. Other techniques for reducing various types of noise in analog transmissions are described in the following sections.

3.4.1 Common-Mode Noise

Common-mode noise refers to noise that is present on both the signal and return terminals of a measuring device such as an A/D module. Common-mode noise may be due to a slight difference in ground potential between the signal source and the A/D module. Whatever the cause, an A/D module with differential input provides a high level of common-mode rejection; that is, it is capable of subtracting out any signal component which is present equally on both the signal and the return lines of the data circuit. This occurs in the following way.

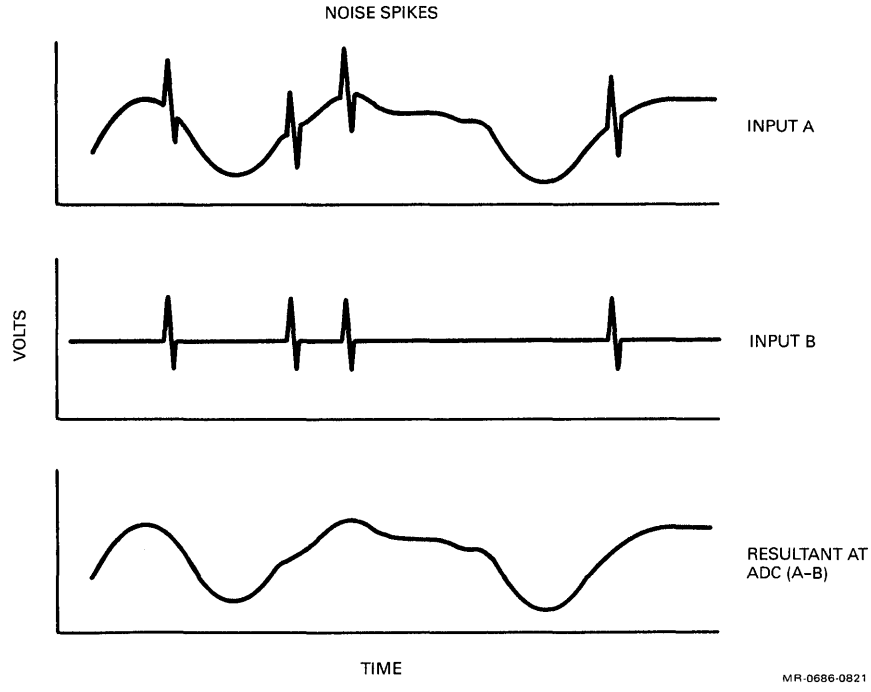
Generally, a data device has two terminals: one positive terminal which produces the data representative voltage, and one minus or return terminal which completes this circuit. Some A/D converters take as input only the positive terminal signal, and measure its potential difference to ground as data (single-ended input). However, other A/D modules take both sides of the data device's signal as input and measure the difference between their respective voltages. Theoretically, this eliminates any noise which is present equally on both signals; that is, common-mode rejection is achieved. For example, if the A/D module is connected as shown in Figure 3-7, any noise present in the environment can be subtracted out. Figure 3-10 shows how any noise (here represented as two spikes) present on both signals is subtracted out. The resulting signal represents the true data more accurately.

3.4.2 Electrostatic Noise

AC voltage sources, such as power lines, lighting, and digital logic components, can become capacitively coupled to signal leads to a degree directly proportional to the combined areas of the radiating and receiving surfaces, and inversely proportional to the distance separating the surfaces. Whether the noise thus coupled to the signal line is objectionable or not depends on a number of factors:

- A low-level electrostatic source has less influence than a high-level one.
- A low-impedance circuit is less influenced by a given level of electrostatic interference than a high-impedance circuit.

Figure 3-10: Common-mode Rejection



- An application attempting to make precise measurements of a 1-millivolt signal is more susceptible to electrostatic noise than one dealing with a 1-volt signal in the same noise environment.

The most effective way of reducing electrostatic noise pickup is to introduce a grounded barrier between the electrostatic source and the signal wire(s). This is done most conveniently by means of shielded leads. Note that shielding is most effective when grounded at only one end, and that it is generally necessary to make such one-sided connections to avoid ground loops. The shield is connected at both ends only when it constitutes the only return path to a floating-source device.

3.4.3 Magnetic Noise

Whenever an electric current passes through a conductor, it generates a magnetic field concentric with the conductor. The strength of this field is directly proportional to the magnitude of the current, and inversely proportional to the distance between the conductor and any reference point. Electric motors, generators, solenoids, and similar apparatus generate magnetic fields of considerable strength.

When any portion of an analog signal wire lies in a magnetic field generated by an alternating current, the field induces an opposing alternating current in the wire. This current flows through the impedances of the circuits to which the wire is attached, and generates noise voltages that are superimposed on whatever signal the circuit normally carries.

The type of shielding that is effective against electrostatic noise has virtually no effect against magnetic noise. The best defense here is the use of twisted-pair conductors. Twisting replaces the single loop created by the signal and return line with a series of small loops. The noise pickup of any given loop in the series tends to be cancelled out by the opposing pickup of neighboring loops. The net result is a significant overall reduction in magnetic noise pickup.

Note that both shielding and twisting can be used simultaneously when both electrostatic and magnetic influences contribute to noise problems.

3.4.4 Crosstalk Noise

Crosstalk occurs when wires carrying unrelated signals lie within close proximity to one another (this happens when multiwire cable is used or when groups of wires are bundled together for some distance.) In such circumstances, the signal in one wire is sometimes coupled by electrostatic and/or magnetic effects into an adjacent wire, and becomes mixed with the signal in the adjacent wire. Crosstalk occurs most frequently when a wire carrying a low-level and/or high-impedance signal lies close to a wire carrying a high-level and/or low-impedance signal.

Cures for the crosstalk problem are like those for electrostatic and magnetic pickup: shielding and twisting either the radiating or the receiving wires—or both, in extreme cases. It is sometimes simpler to isolate sensitive wires (that is, those carrying low-level and/or high-impedance signals) from other wires in the interfacing connections—particularly from wires driving power-consuming devices such as relays or lamps.

3.4.5 Multiplexer Noise

All solid-state multiplexers inject a small amount of charge into their input lines when changing channels. This causes a transient error voltage that is discharged by the source impedance of the input signal. To minimize this problem, it is recommended that all analog signal cables be kept as short as possible. Also, settling errors can be minimized by increasing the time between conversions.

3.4.6 Residual Noise

All electrical circuits generate some random electrical noise within themselves. This inherent noise is called residual noise. The statistically significant effect of residual noise in the A/D module should contribute less than 0.0015% of the full-scale input range to any conversion. Higher preamplifier gains increase the error due to residual noise.

3.4.7 Signal Averaging

Signal averaging is another approach to reducing the effects of noise. Essentially, signal averaging involves performing an application many times and using the average of all the data received as the signal. Signal averaging is based upon two assumptions:

- The data signal is synchronized with some external event .
- The noise effects are random.

By running an application many times, it is assumed that a large number of random noise signals cancel each other out. A positive noise component on one run will cancel out a negative noise component on another run. Increasing the number of runs increases the likelihood that random noise will be cancelled out. In fact, the signal-to-noise ratio of the averaged signal is improved by a factor equal to the square root of the number of runs performed.

Signal averaging is often necessary when the amplitude of the noise component is equal to or greater than the amplitude of the data signal.

3.5 How To Select A Gain Factor

Many A/D modules have intrinsic amplifiers which can be used to boost the input signal, if desired. Such amplifiers are functionally similar to external amplifiers, in that they apply gain to the signal before it reaches the A/D converter. Thus, intrinsic amplifiers can be viewed as dividing the input range of the module by a constant factor. For example, a module configured to accept input signals in the range -10 V to +10 V would appear to have an input range of -1 V to +1 V if a gain factor of 10 were selected. Various modules provide a wide diversity of intrinsic gains, some with very limited range for example, 2X, 4X, and 8X), and others with relatively wide range (for example, 10X, 100X, and 500X).

This section contains a brief description of the operation of fixed- and programmable-gain on-board amplifiers, and a discussion of the special case of autoranging preamplifiers.

3.5.1 Fixed- And Programmable-Gain Amplifiers

The gain of an intrinsic amplifier typically can be set in one of two ways:

- With *fixed gain* modules, the gain factor is selected by setting jumpers on the module. For multichannel modules, the selected gain factor is applied to all channels continuously. To change the gain, the jumper settings must be reconfigured.
- With *programmable gain* modules, the gain factor is selected by setting a bit field in the module's CSR. For multichannel modules, the selected gain factor is applied to all channels until it is changed under program control.

The principle advantage of intrinsic amplifiers is ease-of-use. If an instrument or transducer produced a raw signal which was low in amplitude, you might be able to avoid the necessity of adding another piece of equipment by amplifying the signal right on the A/D module. Furthermore, programmable gain allows you to customize the apparent input range of the module to match the output characteristics of multiple signal sources. This can be done even for individual channels in a multichannel scan, with appropriate software.

The principle disadvantage of intrinsic amplifiers is that they reduce the speed of conversions, since the amplifier must settle before its output to the A/D circuitry is valid. Thus, when a conversion is triggered, the module waits some prescribed interval for the amplifier to settle before beginning the conversion. The length of the settling interval is controlled by on-board circuitry, and is dependent, in part, on the amount of amplification performed. In general, the more gain you select, the slower your conversions will run. Naturally, this factor, like the underlying speed of the A/D converter, is also dependent on the design of the module. The spec sheet of the module should tell you the settling interval at each selectable gain.

Fixed- or programmable-gain levels should be selected to optimally match the input level of the analog signal, while not adversely affecting the speed at which conversions are performed.

3.5.2 Autoranging

Ideally, the optimal gain factor would be selected on a point-by-point basis. When the signal level was low, gain would be applied; when it was high, no gain would be applied. This is exactly what is accomplished using *autoranging*. Some A/D subsystems can operate in a mode wherein the programmable-gain setting of the A/D converter is automatically adjusted according to the amplitude of the input signal.

For example, suppose that the signal generated by a chromatograph detector were being digitized using an A/D converter with an input range of 0 - 10 V. The largest peaks in the signal might be on the order of 8 - 9 V. For these peaks, it would be inappropriate to apply any intrinsic gain, since a gain of even 2X would boost the signal level beyond the input range of the A/D converter. Small peaks, on the other hand, might be on the order of 0.1 - 0.2 V. For these peaks, the voltage resolution of the A/D converter could be optimized by applying a gain of 50X.

Let's examine this case in more detail. Suppose further that the A/D module used a 16-bit converter and could apply gains of 1X, 4X, 16X, and 64X. Table 3-4 shows the optimal gain factor for different signal level ranges and the corresponding voltage resolution of the converted values.

Table 3-4: Programmable Gain

For signals in the range:	the optimal gain setting would be:	yielding a voltage resolution of:
0.000 - 0.156 V	64X	2.4 microvolts
0.156 - 0.624 V	16X	8.6 microvolts
1.250 - 2.500 V	4X	38.4 microvolts
2.500 - 10.00 V	1X	153.6 microvolts

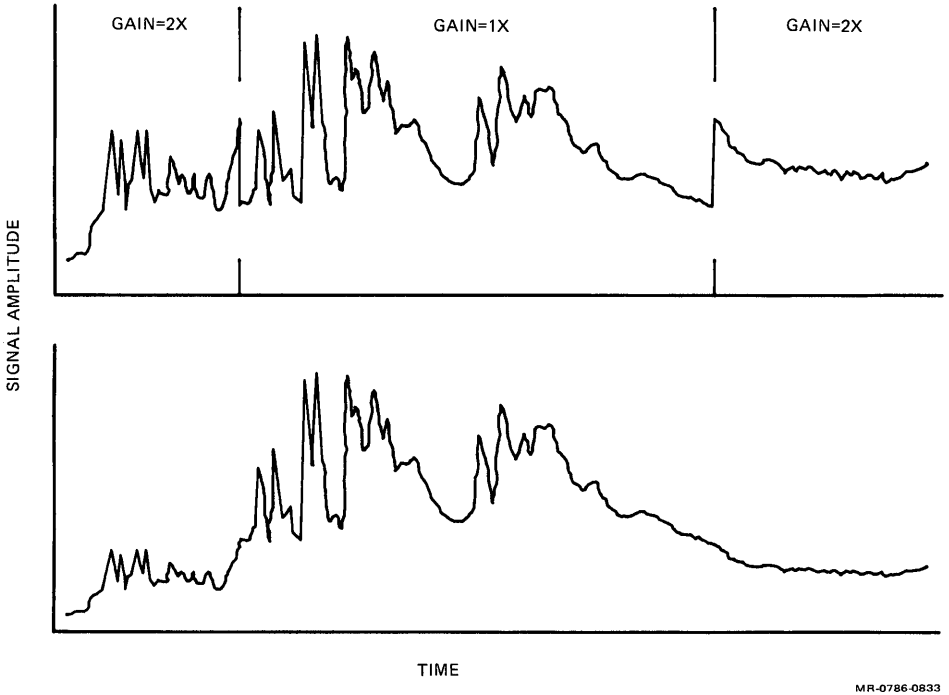
An autoranging preamplifier automatically sets the optimal gain factor depending on the amplitude of the signal at each sample time. In order to properly interpret the converted values, the system needs to know what gain factor was applied for each sample point. One way to accomplish this is by passing an extra byte containing the gain information to the program. Alternatively, some 12- or 14-bit converters pass gain information to the program in the high-order bits of each data value. In either case, when the binary values from the A/D converter are transformed to voltage, the gain information must be decoded and factored into the transfer function.

Chromatography is typical of a class of applications in which autoranging is clearly very useful. The dynamic range of the signals is usually very broad. Low-amplitude peaks near the signal baseline are often of equal interest to high-amplitude peaks near the top of the amplitude range. The amplitude resolution of the signal in the low range must be great enough to adequately characterize these low-amplitude peaks (latency, peak amplitude, integral, etc.). Figure 3-11 illustrates how autoranging affects the digitization of chromatograph detector voltage.

The advantage of using an autoranging preamplifier is that the effective input range of the A/D module is automatically adjusted to maximize the voltage resolution of the system. This can be desirable if the signal contains both high- and low-amplitude components, or if the amplitude range of the signal is unspecified.

As with programmable gain, autoranging slows the speed at which conversions can be performed, because the amplifier settling time becomes a factor when gain is applied. Furthermore, autoranging is only useful

Figure 3-11: Effect of Autoranging

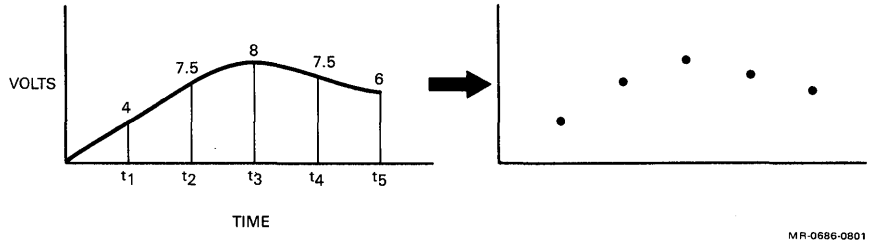


where the signal has a true zero baseline. If the signal has a DC offset or if it exhibits DC drift (as is often the case with chromatograph detector voltages), the "low-amplitude" components of the signal may fall outside the range within which gain can be automatically applied. Finally, autoranging imposes the minor programming constraint that the gain factor applied to each data point must be decoded and applied to any internal numeric conversion formula. For user-written software, this is trivial, but it may preclude the use of an autoranging preamplifier with canned software products not specifically designed with autoranging in mind.

3.6 How To Select A Sampling Rate

As described above (Section 3.1.2), information is often contained in the pattern of changes in signal amplitude over time. To capture these temporal patterns, the signal is typically sampled at discrete, uniform time intervals under the control of a realtime clock. For the digitized signal to accurately represent this temporal information, the sampling rate must be greater than the rate at which significant changes in the amplitude of the analog signal occur. This is illustrated in Figures 3-12 - 3-14.

Figure 3-12: Discrete Time-Interval Sampling (Case 1)



The dots represent the digitized data values. Thus, the computer "knows" only those values. The behavior of the signal between sample times is unknown. In fact, the same sample values might have been generated by the signal shown in Figure 3-13.

In order to accurately represent the signal shown in Figure 3-13, a higher sampling rate is required, as shown in Figure 3-14.

Figure 3-13: Discrete Time-Interval Sampling (Case 2)

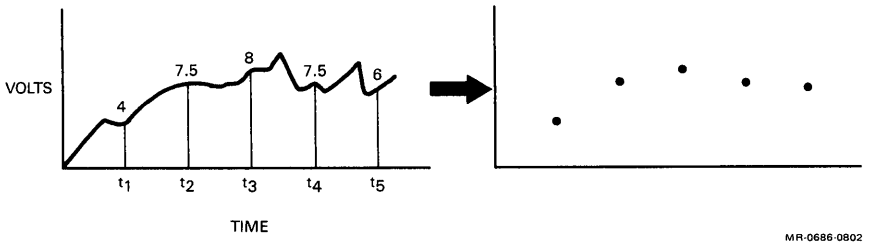
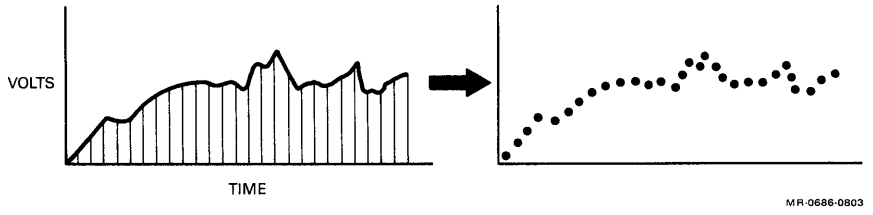


Figure 3-14: Discrete Time-Interval Sampling (Case 3)



The optimal sampling rate is the lowest rate which still preserves the information content of the signal. Naturally, you can insure that as much information as possible is retained by always sampling at the highest rate of which the computer system is capable. In most cases, though, this only strains the resources of the system, and does not add meaningfully to the information content of the digitized signal. In the following sections, the factors which determine the optimal sampling rate are discussed.

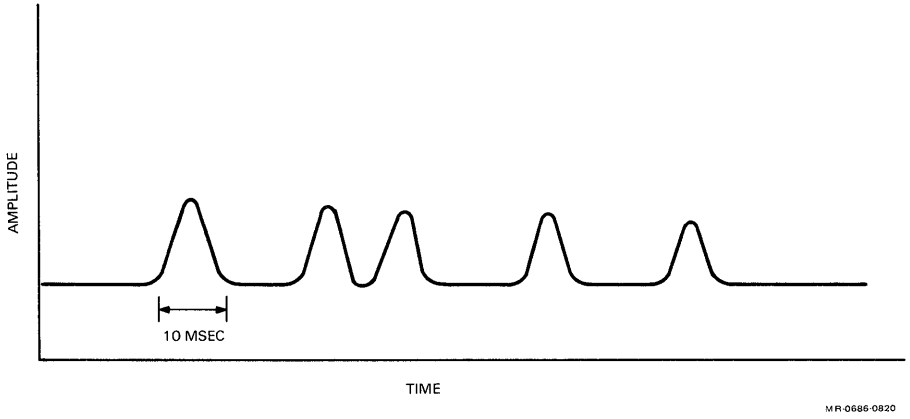
How you determine the optimal sampling rate depends on how you intend to analyze the data. Time-varying signals can be analyzed in either the time-domain or the frequency-domain.

3.6.1 Time-Domain Analyses

In time-domain analyses, one is typically concerned with questions like "When did a relative maximum occur?," "What was the amplitude at the maximum?," or "What was the duration of the event?" To insure that these sorts of information are preserved in a digitized signal, you must select a sampling rate that provides adequate resolution in the time domain. How much resolution is required depends on what information you are trying to extract.

Consider, for example, the signal shown in Figure 3-15. The width of each pulse is 10 mSec at its base. By digitizing the signal at any sampling rate greater than 100 Hz, you could be certain that at least one data point for each pulse was above the baseline. If the goal of your analysis were to detect simply the presence of voltage transients, a sampling rate of, say, 110 Hz would suffice. You would be assured of detecting any pulse.

Figure 3-15: Example of a Signal for Time-Domain Analysis



If your goal were to count peaks, you would have to set the sampling rate high enough so that a relative maximum would be apparent for each peak. Since there is no overlap of peaks in the example shown, a sampling rate of 200 Hz would suffice to insure that there would be a relative maximum in the digitized data for each peak in the analog signal.

If your goal were to determine the time at which each peak occured (peak latency) the sampling rate should be set according to the required degree of accuracy. At a sampling rate of 500 Hz, the accuracy would be +/- 1 mSec. The general formula for resolution is:

$$\text{latency resolution} = +/- \ 1/(2R)$$

where R is the sampling rate in Hz.

3.6.2 Frequency-Domain Analyses

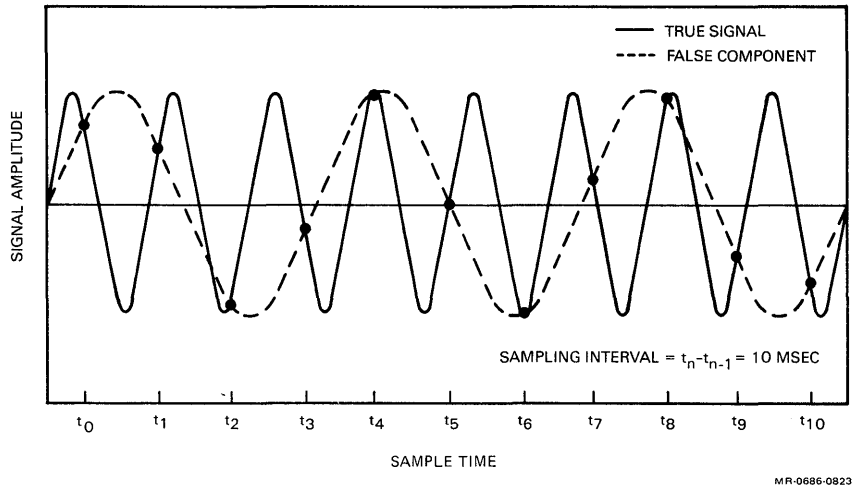
Many types of frequency-domain analysis are based on Fourier analysis. The fundamental principle of Fourier analysis is that any signal can be decomposed into sinusoidal components, each of which is characterized by its frequency, amplitude, and phase. When the component sinusoids are algebraically summed, the original waveform is reproduced. If the signal consists of discrete time samples, as is the case with digitized data, Fourier theory states that the signal can be characterized by N/2 sinusoidal components, where N is the number of sample points.

Take, for example, the case of a signal digitized at a sampling rate of 100 Hz for 1 second (Hz = samples/sec). By applying the Fourier transform, the signal could be decomposed into 50 sinusoids with periods which were integral dividends of the total sampling interval - 1 second. That is, the sinusoids would have periods of 1/1, 1/2, 1/3, ... 1/49, and 1/50 seconds. Each of these frequency components would have an associated amplitude and phase. The sum of all 50 components would equal the original signal. The highest of these frequency components, 50 Hz in this case, is called the *Nyquist frequency*.

Stated differently, the highest frequency component which can be derived from the Fourier transform of a digitized signal is N/2P Hz, where P is the total sample interval and N is the number of sample points. This implies that, if you want to use the Fourier transform to evaluate the frequency content of a signal, THE SAMPLING RATE MUST BE AT LEAST TWICE AS FAST AS THE HIGHEST FREQUENCY OF INTEREST IN THE SIGNAL. If you selected too low a sampling rate, the Fourier transform of the digitized signal would not contain the information you wanted to extract. Moreover, if the raw signal contained frequency components above the Nyquist frequency, those components would be improperly transformed by virtue of a phenomenon called "aliasing."

Figure 3-16 shows the effect of aliasing on digitization of a hypothetical sinusoidal signal. The raw signal is shown as a solid line. In this example, the frequency of the signal is 80 Hz (16.7 mSec period), and the sampling rate is 100 Hz (10.0 mSec sampling interval), yielding a Nyquist frequency of 50 Hz.

Figure 3-16: The Phenomenon of Aliasing



The sampled data points are equally well fit by a 30 Hz sinusoid, shown as a dotted line. The Fourier transform of the sampled data would show a power peak at 30 Hz when, in fact, no such component was present in the raw signal. To avoid this problem, you must insure that the Nyquist frequency, determined by the sampling rate, is higher than any major frequency components of the raw signal. If the signal contains frequency components above the Nyquist frequency which are not of any experimental interest (i.e., noise), you should either raise the sampling rate to prevent aliasing, or filter those components out of the analog signal prior to digitizing.

Another constraint of many Fourier transform algorithms is that the number of data points must be an integral power of 2; 256, 512, 1024, etc. Though this limitation does not explicitly constrain the sampling rate, you should consider its implications. If you wanted to achieve a Nyquist frequency of, say, 500 Hz, you would set the sampling rate at 1000 Hz. In order to conform to the constraint on sample size, you should then collect some number of data points that was an integral power of 2, say 2048. That means that your total sample interval would be 2.048 seconds. When you applied the Fourier transform to those data, the frequencies of the component sinusoids would be 1/2.048, 2/2.048, 3/2.048, ... 1023/2.048, and 1024/2.048 Hz. It might have been better to select a sampling rate of 1024 Hz, yielding a Nyquist frequency of 512 Hz. You could then meet the constraint on numbers of data points by collecting data for 2 seconds, and the Fourier transform would yield frequency components of 1/2, 2/2, 3/2, 4/2, ... 1023/2, and 1024/2 Hz. Depending on how you intend to make use of the output of the Fourier transform, it can be desirable to select a sampling rate to yield frequency components at intervals which are simple fractions of cycles-per-second.

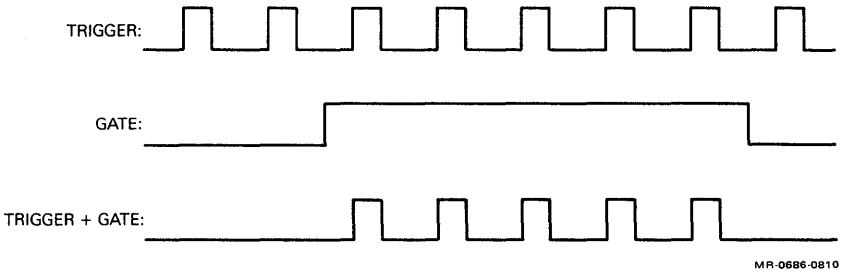
3.7 How To Select A Trigger Mode

In order for digitized data to be meaningfully interpreted, the sampling of the analog signal must be synchronized either with some external event or with a timebase. In some cases, both types of synchronization are required. For example, in physiology, evoked potentials might be studied by digitizing the electrical signal coming from a nerve in the arm at repeated 100 μSec intervals following the application of a mild shock to the hand. Each A/D conversion would be synchronized with a 10 kHz timebase while the timebase itself would be synchronized with the onset of the shock. This process of synchronization is called triggering. Unfortunately, the term "trigger" is used somewhat loosely to designate several kinds of events and operations. Before proceeding, then, several basic concepts will be defined.

3.7.1 Triggers And Gates

A *trigger* is an event or signal which causes some sequence of operations to be initiated. Although trigger signals are often pulses of some finite duration, conceptually they can be viewed as level transitions or edges. That is, the duration of a trigger signal is of no interest; only its onset is meaningful. This is in contrast to the concept of a *gate*, which is a signal whose level controls the passage or blockage of trigger signals. The relationship between trigger and gate signals is illustrated in Figure 3-17.

Figure 3-17: Triggers and Gates



Both trigger and gate signals can originate either internally or externally with respect to the computer system. Internal trigger or gate signals could come from a realtime clock module, or be generated by software (that is, by the setting or clearing of bits in the device's control/status register). External trigger or gate signals might be derived from switch closures or sync pulses generated by control logic, oscilloscopes, and the like.

3.7.2 Timebase Triggering

Discrete time-interval sampling, as discussed in Section 3.1.2., involves triggering A/D conversions at repeated, constant intervals. Time intervals can be generated by either an external oscillator of some kind or a realtime clock module. Thus, the output of the clock or oscillator serves as the trigger signal for A/D conversions. Each clock pulse triggers a single sample. The clock or oscillator signal can be led to the A/D module via connectors on the outer edge of the modules or, in the case of a clock residing on the system bus, via a bus rail. As described in the preceding section, the clock pulses can be gated by another signal, or through software.

But what controls the clock? How is its operation synchronized with external events? The answer is that realtime clocks and oscillators can, in turn, be controlled by trigger and gate signals. At this point, it becomes important to discriminate between A/D trigger events and clock trigger events. Like A/D triggers and gates, clock triggers and gates can originate either internally or externally to the computer system.

3.7.3 Common Trigger Modes And When To Use Them

Internal and external gates, triggers, and clocks can be put together in an astonishing number of configurations. In the following paragraphs, brief descriptions of some of the most common of these combinations are given. For the sake of clarity, it is assumed that only one channel is to be sampled. Triggering of multichannel scans is described in Section 3.7.4.

Software triggering is the simplest form of A/D triggering. In this mode, a single A/D conversion is initiated by setting the appropriate bits in the A/D CSR register. Upon completion of the conversion, the single data value is available in the buffer register of the A/D module. The software sequence for initiating the conversion can be synchronized with the system clock or with a command entered at the console. This mode is useful when only very loose synchronization is called for and only a single data value is needed. Software triggering might be used, for example, to obtain a temperature reading every five minutes, or periodically at the operator's discretion.

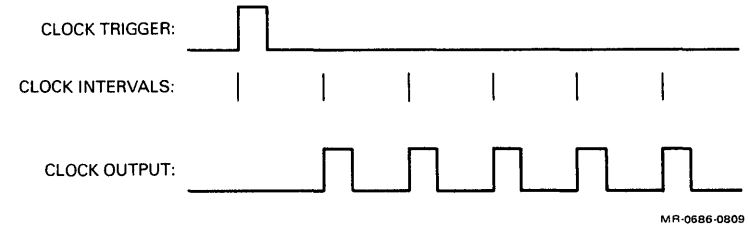
External triggering is like software triggering in that a single sample is taken on each trigger. In this mode, however, the trigger signal is generated by some external event - for example, the interruption of a photobeam. The latency between the onset of the trigger signal and the time at which the A/D conversion begins is typically very short and highly reproducible, since triggering is handled completely in hardware by the A/D module. Note, though, that the rate at which multiple externally triggered conversions can be handled by the system is dependent on how fast the data can be moved into memory, either under program control or via DMA. This mode of triggering is used where tight synchronization of A/D sampling with an unpredictable external event is desired. It might be used, for example, in measuring the weight or albedo of parts moving through an assembly line.

Triggered timebase triggering differs from the preceding two modes in that the A/D conversions are triggered by the output of a clock, either internal or external. The clock, in turn, is triggered by some external signal. This mode of operation is analogous to the triggered sweep mode of an oscilloscope. The external trigger event effectively starts a series of A/D conversions, each of which is triggered by a clock output pulse. Commonly, clock output signals continue to trigger A/D conversions until either a second clock trigger event shuts off the clock or a predetermined number of conversions have been performed.

In the latter case, the total sampling interval (that is, the time interval which encompasses all A/D samples) can be determined by multiplying the clock interval by the number of samples. This mode of triggering is used when a "time-sweep" of data, synchronized with some external event, is desired. As described in the preceding sections, time-sweeps of data are used when information is contained in the temporal variations of the signal, rather than in the instantaneous signal level. For example, in an automobile crash test, clocked A/D conversions might be triggered by the breaking of a photobeam several feet from the crash wall.

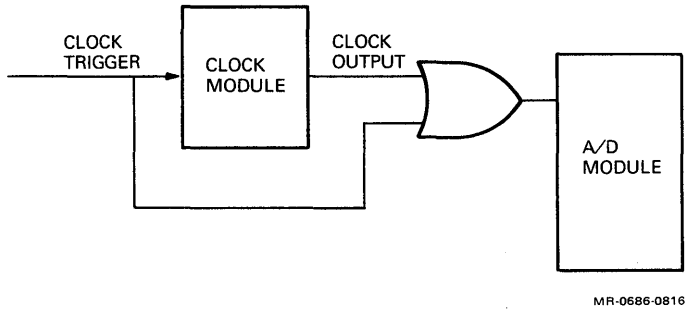
It should be noted that many clock modules do not produce the first output pulse until one clock interval after the occurrence of the trigger event. This is illustrated in Figure 3-18.

Figure 3-18: Triggered Timebase Triggering



If you use a clock module which has this feature, the first A/D conversion in a time-sweep will not be triggered at the time of the clock trigger, but rather at a latency of one clock interval. In most applications, this offset poses no problem if it is properly accounted for in subsequent analyses. If your application requires that the first A/D conversion be triggered simultaneously with the triggering of the clock, special circuitry may be required, as illustrated schematically in Figure 3-19.

Figure 3-19: Alternative A/D Triggering Circuit



Gated timebase triggering can be viewed as a variant of triggered timebase triggering. As above, each A/D conversion is triggered by the output of a clock which is synchronized with an external event. In this mode, however, the external event is used as a gate, rather than as a trigger. Thus, the clock begins generating A/D trigger signals when its gate signal is asserted and continues to run until the gate signal is deasserted. The underlying clock output can be free-running, or it can be synchronized with the onset of the gate signal. In the free-running case, the latency from the assertion of the gate signal to the occurrence of the first A/D trigger is unknown, though it is never more than one clock interval. In the synchronized case, the latency from the assertion of the gate signal to the first A/D trigger is equal to one clock interval.

This triggering mode is used when the length, as well as the onset, of the sampling interval is to be controlled externally. For example, in liquid chromatography, it is necessary to sample data on a constant timebase for the entire period that the pump is on. However, the time required to elute all the analytes of interest may vary from one run to the next. By gating clock output with the pump control signal, the desired result is obtained. This mode of triggering can be used to synchronize an A/D time-sweep with an oscilloscope trace. By using the oscilloscope gate-output to gate the A/D timebase, the total sampling interval can adjust automatically to any changes in the oscilloscope's timebase.

Triggered timebase triggering with prestimulus sampling is used to perform clocked A/D sampling in some period prior to the occurrence of a trigger event. Obviously, to accomplish this, A/D sampling must be continuously ongoing before the trigger event occurs. Data are stored in a circular buffer, either on the A/D board or in system memory. When the trigger event occurs, a pointer to the current location within the circular buffer is saved, and data acquisition continues uninterrupted. Subsequently, data collected in advance of the pointer can be taken from the circular buffer and stored, along with data following the pointer, if desired.

This mode of triggering is used when it is necessary to collect data prior to the occurrence of some uncontrollable event. For example, the membrane potential of a neuron prior to the occurrence of spontaneous spikes could be studied using this triggering mode. The clock trigger could be derived from the spike, and data acquired for several milliseconds prior to spike onset.

Note that this triggering mode imposes considerable overhead on the system, due to the complicated buffer management scheme required. This not only slows down the system, but requires additional programming effort as well. Thus, this triggering mode should be avoided in favor of more direct modes, where possible. If the trigger event can be controlled by the experimenter, it may be possible to implement a circuit which generates a clock trigger signal prior to the desired synchronizing event. In the example given above, it might be possible to generate a clock trigger, followed at some latency by an electrical stimulus to the neuron which caused it to fire. Though this would be much better for A/D triggering, the spontaneous nature of the neural firings would be lost.

3.7.4 Multichannel Scanning

Sampling multiple channels on a single trigger is called *scanning*. Scans of multiple channels can be triggered in all of the modes described above. Operationally, the only difference is that several A/D conversions, rather than only one, are performed following each A/D trigger. Data values from all channels sampled on a single trigger are usually, though not always, treated as having been sampled simultaneously (or nearly so). That is, there is usually an intent to characterize or analyze the temporal correlation between activity on the different channels, either explicitly or implicitly. Some A/D modules, particularly DMA modules, have provision for specifying how many and which channels will be sampled following each A/D trigger. For boards which do not have such a feature, channel scanning must be implemented in software. In either case, multiple channels sampled on a single trigger are not sampled truly simultaneously unless the A/D board has sample-and-hold circuitry for each input channel.

The difference in sampling times between channels is called *skew*. When a trigger event occurs, each channel is sampled in sequence at the maximum rate of which the A/D converter is capable. For example, if a 50 kHz A/D board were used to scan 3 channels, the first channel would be sampled at the time of the A/D trigger event, the second channel would be sampled 200 μSecs later, and the third channel would be sampled 400 μSecs after the trigger. In most applications, skew is either inconsequentially small or else it can be accounted for in analysis. For example, if four analog signals were being sampled at a rate of 100 Hz with a skew of 200 μSecs, the maximum channel skew (600 μSecs) would be 6% of the timebase interval. If this degree of inaccuracy were unacceptable, sample-and-hold circuitry would be called for.

inaccuracy were unacceptable, sample-and-hold circuitry would be called for.

The overall rate at which an A/D module can function is dependent on the number of channels being scanned, since the A/D converter digitizes only one channel at a time (even if per-channel sample-and-hold circuitry is used). For example, an A/D module rated at 50 kHz can scan 5 channels at a rate of 10 kHz, theoretically. The speed of the timebase multiplied by the number of channels scanned is called the *aggregate sampling rate.* In selecting an A/D module, it is important to bear in mind that the rated speed of the board is the maximum aggregate sampling rate for multichannel scanning.

Chapter 4
How To Perform Analog Output

The problem of producing or controlling analog voltage levels with a digital computer is, in many respects, similar to that of converting analog signals to digital values. Many of the same concepts apply, such as the discrete nature of the signal in both the amplitude and time dimensions, numeric representation of voltage levels, and module configuration for signal range and polarity. If you have not read the chapter on analog input, you should do so before reading this chapter.

4.1 Analog Output Modules

Analog output modules differ from analog input modules in one fundamental respect - the channels are not multiplexed. As described in the preceding chapter, the following sequence of steps is performed to read from one channel on an analog input module:

1. The desired channel number is placed in the appropriate bit-field of the CSR of the A/D module.
2. A conversion on that channel is started by setting the GO bit in the CSR or by the occurrence of a trigger event.
3. When the conversion is complete, the program reads the converted data value from the module's single A/D buffer register.

With multichannel D/A modules, each channel has a dedicated buffer register. To output an analog voltage on any given channel, the program simply loads the buffer register for that channel with the desired digital value. (For DMA modules, the same principle holds, but the output buffer registers are loaded directly from memory by the module's DMA controller.) Each output buffer register, in turn, is "hard-wired" to a dedicated D/A converter.

The reason for this difference between analog input and output modules is clear when you consider the nature of the two types of conversion. On input, when a data value is requested, the voltage on the selected channel is latched before conversion. The converted value represents the voltage level on that channel at a single moment in time. However, on output, the converted value (that is, the voltage) is meant to remain asserted until the program changes it - possibly indefinitely. This can only be achieved if the digital value to be converted remains valid for the entire duration of the desired output. Hence, each analog output channel must have its own dedicated buffer register-D/A converter pair.

An important side effect of this arrangement is that the speed of conversion in an analog output module is not limited by multiplexer switching or settling time. Once the buffer register is loaded, the latency to the appearance of a valid voltage on the associated output line is limited only by the speed of the D/A circuit. This is typically very fast, relative to the time required to perform an analog input operation. Note, though, that it is NOT equivalent to the maximum rate at which you can load the buffer register with new values. The speed of operation of a D/A converter is typically expressed in terms of the time required for the voltage output to reach 99.9% of the final level for a full-scale output change. For example, the AAV11-D is rated at a settling time of 5 microseconds for a full-scale step of 10 volts. The settling time for a 100-mvolt step is 1 microsecond. Thus, the AAV11-D output can theoretically be driven at 200 kHz with large voltage steps and at approximately 1 MHz with small voltage steps.

Since analog output is intrinsically less complicated than analog input - no gain is applied, no multiplexing is performed - some analog output modules have no control/status register. One of the functions of the CSR on an analog input module is triggering. Analog output on modules which lack a CSR is "triggered" by simply loading the appropriate output register. Analog output modules with fuller functionality, particularly DMA modules such as the AAV11-D, have a CSR through which support for an external trigger is provided. Modules which support external

triggering provide mechanisms for the program to detect the occurrence of a trigger, either via a status bit in the CSR or via interrupts.

In addition to the analog output channels, some analog output modules provide several digital output lines which can be used to synchronize the operation of an external device with the analog output module. For example, the AAV11-C and AAV11-D modules have four such lines. One important application for these lines is to control the ancillary functions of an oscilloscope - blank, unblank, erase, and intensity (z-axis). On the AAV11-D these bits are controlled by the program through a bit-field in the CSR. On the AAV11-C, they are controlled through the low-order 4 bits of one of the output buffer registers.

4.2 How To Connect The Option Module To The External Device

Analog output voltages and digital control signals (if present) leave the module via an edge connector on the top of the board. Cabinet kits are available for most modules to lead the signal lines to a bulkhead connector. Each analog output line has an associated return line for connecting the signal to devices requiring differential input. For single-ended output, connect the return lines to circuit ground on the external device. DO NOT strap unused analog output lines to circuit ground, as this may overload the D/A converter circuits.

4.3 How To Select A Trigger Mode

Any of the triggering modes described in Section 3.5.3 can be used on analog output, with the exception of triggered timebase triggering with prestimulus sampling. On modules which have an external trigger control, a clock output pulse can be strapped to the external trigger input to initiate D/A conversion on clock overflow without program intervention. In this case, the output buffer must be loaded with the desired value prior to each clock pulse. Modules which do not have an external trigger input must be "triggered" under software control by loading a value into the appropriate output buffer for immediate conversion. In this case, synchronization of analog output with an external event or timebase must be accomplished by polling or using interrupts generated by another device, such as a clock/counter module.

Commonly, DMA analog output modules support external triggering, while PIO modules do not. This allows a DMA module to perform timebase-driven analog output without program intervention. The AAV11-D also allows the user to perform 2-channel DMA output on external trigger. The data values from memory are output on the two channels alternately, one pair of values per trigger event. When operating in this mode, the user program must interleave the data values for the two channels in memory to ensure proper operation.

Chapter 5
How To Perform Digital I/O

Digital signals are used to represent information that is binary in nature. Examples of binary information are switch position (open or closed), indicator light status (on or off), and the status of individual bits in a binary data word (set or clear). Like analog signals, digital signals represent information in the form of voltage levels. However, only two discrete voltage levels are needed to represent each unit of binary information. In contrast, analog signals can assume any voltage level within some finite range.

In the most primitive sense, digital information can be represented by any two voltage levels, as long as the two levels are discriminably different. In practice, digital information is represented by certain conventional voltage levels. These conventions define not only the voltage levels corresponding to the on and off states, but other characteristics of the electrical circuits used to produce and sense the voltages as well. Common digital conventions are transistor-transistor logic (TTL), emitter-coupled logic (ECL), and CMOS logic. Of these, TTL is by far the most widely used for digital I/O modules. In the remainder of this chapter, assume that the digital signals discussed are TTL signals, except where otherwise noted.

With TTL, signal levels between 0.0 (ground) and +0.8 volts are considered "low" and signal levels between +2.4 and +4.5 volts are considered "high." Some devices can accommodate signal levels outside this range, for example, interpreting any signal greater than +2.0 volts to be

high. In general, though, you should avoid using signal levels outside the standard TTL ranges, if possible.

Though TTL-low and TTL-high levels are reasonably standardized, their interpretation is not. Some devices interpret TTL-high levels as signalling an active or *asserted* state, while others interpret TTL-low levels as asserted. These interpretations are called positive- and negative-TTL logic, respectively. Obviously, the informational meaning of a TTL level depends on whether the device uses positive or negative logic. In most cases, troublesome differences between devices in TTL logic polarity can be accommodated by appropriate program logic. In some cases, however, it may be necessary to build external circuitry to invert the signal level. In any case, you should be aware of the logic polarity of any device which uses TTL logic.

5.1 Digital I/O Modules

Like other types of modules, digital I/O modules have one or more control/status registers and two or more data buffer registers which appear as storage locations in the computer's I/O page. At a minimum, there is one control/status register, one data buffer register for input of data from external devices, and one data buffer register for output of data. DMA modules also have registers for controlling direct-memory-access transmissions. On the external side of the module, there are minimally 16 input data lines, 16 output data lines, and 2 handshake lines for synchronizing data transmissions in each direction (for a total of 4 handshake lines).

The operation of digital I/O modules differs widely among module types. For example, the DR11-C has little more than the minimal functionality described in the preceding paragraph. The DRV11-J has four 16-bit wide ports, each of which can be configured for either input or output. Both these modules support programmed I/O only. The DRV11-WA is a DMA module which has two 16-bit wide ports, one for input and one for output. In addition to the minimal handshake lines, the DRV11-WA has 6 additional device synchronization lines (3 input and 3 output) for implementing user-defined functions.

In the laboratory, two major classes of applications for digital signals are found:

- The term *discrete digital* is used to describe the use of digital logic levels to control or detect relatively independent events.
- The term *parallel digital* is used to describe the use of digital logic levels to transmit formatted data between two devices.

The use of TTL signals for solenoid control is an example of discrete digital I/O. For example, 16 digital output lines might be connected to individual solenoid drivers. Which solenoids were thrown and which were open at any time would depend on the state (TTL-high or TTL-low) of each of the 16 signal lines. The state of any given line and its associated solenoid might be changed asynchronously of the other lines.

Parallel digital I/O is used for formatted data transfers. The major types of formatted data are 8-bit binary, 16-bit binary, and binary-coded decimal (BCD). With formatted data, information is contained in the pattern of TTL-low and TTL-high levels across several lines, rather than in the state of each individual line. Changes in TTL levels occur synchronously across all the lines carrying the formatted data.

The distinction between discrete and parallel digital I/O has important consequences for the programmer. Discrete digital I/O is usually performed using programmed I/O (PIO), whereas parallel digital I/O often uses direct memory access (DMA). Discrete digital data is often "masked" so that only certain lines are controlled or tested at any one time, whereas parallel digital data is handled in full bytes or words. Finally, discrete digital I/O is usually event- or timebase-driven, whereas parallel digital I/O involves some form of handshaking protocol for device synchronization.

These differences notwithstanding, many factors regarding signal conditioning, cabling, and so forth are common to both discrete and parallel digital I/O. These common factors are discussed in Sections 5.2 and 5.3. Operational and programming considerations unique to each type of digital I/O are discussed in Sections 5.4 (discrete) and 5.5 (parallel).

5.2 How To Condition Digital Signals

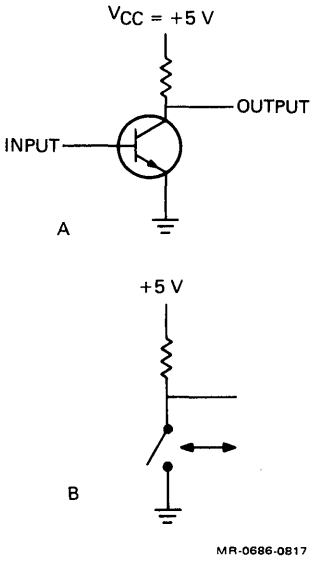
In this section, the electrical properties of TTL circuits are described in more detail. TTL conventions constitute a fairly complete definition of signal levels and the properties of circuits for producing and sensing the signals. Devices which use standard TTL signals often can be connected directly to a digital I/O module without the need for any additional signal-conditioning circuitry. If the "device" is a switch, the position of which must be sensed by the program, the user must generate the signal by implementing a simple circuit. If the device is a solenoid, additional circuitry may be needed to convert the TTL signal from the digital I/O module into a higher power signal.

5.2.1 Properties Of TTL Circuits

Any TTL device or circuit, including a port on a digital I/O module, functions as either a source (transmitter) or as a receiver of TTL signals. Some circuits can function as both, in the sense that either the transmitter or the receiver portion of the circuit can be active at any given time. For example, the ports of the DRV11-J module may be configured as either output (transmitter) or input (receiver) ports, depending on the setting of certain bits in the device's CSR.

Figure 5-1A shows a simple TTL transmitter circuit. The transistor can be thought of as loosely equivalent to a mechanical switch, as shown in Figure 5-1B. Changing the current applied at the input of the transistor changes the impedance (resistance) of the path between Vcc and ground. When the impedance of the transistor is very low, the output is effectively tied to ground (as though the switch in 5-1B was closed), and the circuit transmits a TTL-low signal. When the impedance of the transistor is high, the output is effectively tied to Vcc through the resistor (as though the path to ground was broken by opening the switch), and the output transmits a TTL-high signal.

Figure 5-1: Transistor-Transistor Logic (TTL) Circuit



A TTL receiver circuit can look very much like the transmitter circuit shown in Figure 5-1A, except that the definitions of the input and output lines are reversed.

TTL transmitters and receivers are well defined with respect to certain circuit properties. The TTL convention defines not only the voltage levels associated with the high and low states, but also the amount of current that flows in each state. A receiver circuit which draws 40 μA of current from the transmitter when its input is high, and which delivers 1.6 mA of current to ground through the transmitter when its input is low, is said to constitute one TTL *unit load*. TTL transmitter circuits are rated in terms of the number of unit loads they can drive. If more than one receiver is connected to a single transmitter, the combined (added) unit loads of all the receivers must not exceed the drive capacity of the transmitter.

Additional circuitry, called *line-drivers*, must be added if the application calls for more transmitter drive capacity than is supplied by the device or digital I/O module. A line-driver circuit can be thought of as simply another transmitter circuit, perhaps like that shown in Figure 5-1A, interposed between the original transmitter and the receiver. The receiver side of a line-driver circuit usually constitutes one TTL unit load, while the transmitter side usually has a drive capacity of 20 or more unit loads. TTL line-drivers are commercially available in DIP packages of eight or more drivers each.

Note

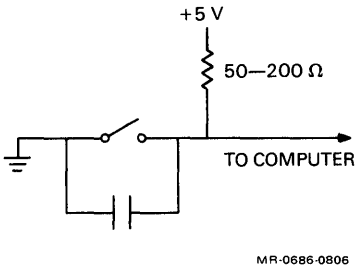
Most TTL line-driver circuits have the effect of inverting the polarity of the signal, in addition to boosting the current. If line-drivers are used, this factor must be considered when programming the computer.

Some TTL transmitter circuits can assume not two, but three, operational states: high, low, and OFF. In the OFF state, the transmitter circuit is functionally disconnected from the line (imagine the line physically disconnected from the circuit). TTL circuits capable of this mode of operation are called *tri-state drivers*. Tri-state drivers are used when multiple drivers are connected in parallel on each line, as in the case of a bus. This is necessary because tying the outputs of two active transmitters together can damage the circuits or cause them to function improperly. Tri-state drivers are also used in devices which can be configured as either transmitters or receivers, such as the ports of the DRV11-J. When the port is a receiver, the transmitter circuits on each line are set in the OFF state. When the port is a transmitter, the receiver circuit is isolated from each line (in the OFF state).

5.2.2 How To Generate TTL Levels From Switch Closures

A common use of discrete digital (TTL) signals in the laboratory is to sense switch closure. The circuit shown in Figure 5-2 can be used to supply TTL-high or TTL-low signals, depending on the state of a switch. The capacitor is optional and is used to reduce switch bounce, if necessary. The capacitor provides a low impedance pathway to ground for high-frequency components of the signal (that is, switch bounce). This has the side effect of slightly increasing the latency between the time at which the switch is closed and the time at which the line assumes the TTL-high state. If you must de-bounce the switch, choose the smallest-valued capacitor which provides the required effect. This can be determined by monitoring the signal on an oscilloscope while trying different values of capacitance.

Figure 5-2: TTL Level from a Switch Closure



5.3 How To Connect Devices To The Digital I/O Module

In theory, the cable connecting two digital devices is merely a passive channel for signals. In practice, each wire in a cable acts like a low-valued series capacitor. Cable capacitance is dependent on the gauge of the conductor wires, and is usually given as one of the specifications by the cable manufacturer. Furthermore, each conductor of a multiconductor cable can act as a weak antenna and pick up signals being broadcast by other conductors in the same cable. This section describes some of the problems that can arise as a result of the active properties of cables, and presents some possible solutions.

5.3.1 Types Of Cables

The two main classes of multiconductor cables used to lead digital signals between devices are bundled cables and ribbon cables. In bundled cables, the conductors are gathered into a round bundle; in ribbon cables, the conductors are strictly side-by-side, resulting in a flat cable.

Either type of cable may have a shield surrounding all conductors. In some bundled cables, pairs of conductors are twisted together. Each twisted pair may be enclosed in a mylar shield. When grounded, the mylar shield forms a barrier against extrinsic signals of all frequencies. This type of shield is sometimes called a radio-frequency (RF) shield.

The common shield which encases all the conductors in a bundled cable is often a braided shield. Braided shields form a barrier only to fairly low-frequency extrinsic signals, including 60-cycle noise. In electrically noisy environments, you should use RF-shielded cables to exclude high-frequency artifacts caused by sources of electrical noise such as switched motors (on air conditioners and refrigeration units, for example).

5.3.2 How To Control Cross-Talk

The term *cross-talk* refers to the tendency for one conductor to pick up signals broadcast by other conductors in the cable. If you have a problem of cross-talk, lines that should be read as TTL-low sometimes appear high (high lines never appear low). In parallel digital transmission, certain bit patterns may be always transmitted incorrectly because of severe cross-talk between particular combinations of conductors.

Though the antenna properties of the conductors are very weak (if TTL circuits are used), the fact that the conductors are in close proximity to each other, often over long distances, exacerbates cross-talk. The simplest way to control cross-talk is to use the shortest possible cable. If this is insufficient, or if a long cable is required, the conductors must be isolated from each other in some way.

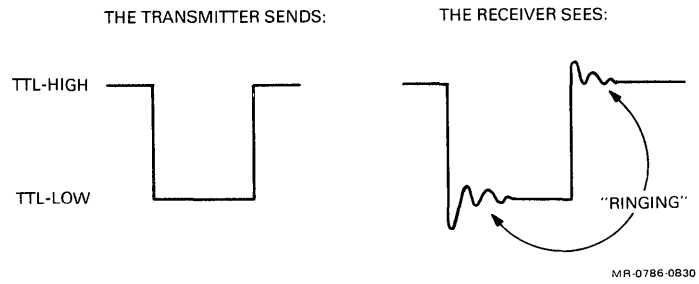
By using a ribbon cable, the number of possible sources of cross-talk for any one conductor is minimized, since the antenna effect diminishes rapidly as the distance between conductors increases. This advantage can be maximized by tying every other conductor in a ribbon cable to ground so that no conductor is adjacent to another active conductor.

Cross-talk can be minimized in bundled cables by using individually-shielded twisted-pair cables. By grounding one conductor of each pair, cross-talk can be virtually eliminated, since each signal line is then individually enclosed in an RF shield. Unfortunately, this solution is cumbersome, since such cables are heavy and stiff.

5.3.3 How To Control Ringing

The term *ringing* refers to the tendency for signal lines to exhibit damped oscillations in response to a step change in signal level. This effect is illustrated in Figure 5-3.

Figure 5-3: Ringing in a Digital Signal



Ringing is caused by a combination of factors, including impedance mismatching between the cable and the receiver, reflectance of the signal back through the cable, and cable capacitance. Thus, cable length, capacitance, and resistance all influence the amplitude and duration of ringing. If the ringing is severe, improper transmission may result. For example, following the high-to-low transition shown in Figure 5-3, if the first positive-going swing brought the signal level above 0.8 volts, the receiver might interpret this as a second TTL-high signal.

Most applications can accommodate some ringing, since the duration of the oscillations is usually very short in comparison to the total data cycle time. However, ringing could be problematic if, for example, the receiver were connected to a flip-flop whose state changed on each positive-going transition.

If ringing is a problem, one simple solution is to put a 20 - 50 Ohm resistor in series with each conductor on the transmitter side of the cable. This slows the rate at which the cable charges (remember, it's a capacitor). Too large a resistor may slow the signal down to an unacceptable degree or cause the signal amplitude to drop too low.

5.4 How To Perform Discrete Digital I/O

As described in the first section of this chapter, discrete digital I/O involves the handling of TTL signals to or from devices such as switches, indicator lamps, photo-beam position detectors, and solenoids. Often, though not always, each line constitutes a discrete unit of information. For example, a behavioral testing experiment might entail monitoring an animal's activity in a maze by sensing photo-beam interruptions and intermittently raising or lowering sliding doors depending on the derived information about the animal's position in the maze. Indeed, several mazes might be operated simultaneously by a single computer system. In this example, the activity on individual lines is largely independent of activity on other lines.

5.4.1 Event-Driven Vs Timebase-Driven Discrete Digital I/O

In cases such as the maze-behavior example, the flow of information and, consequently, the sequence of program execution is event-driven. Each time a new photo-beam is broken, the program logs the information, invokes a decision-making algorithm, possibly raises or lowers one or more doors, then waits for the animal's next move. This sequence is triggered by an external event - the breaking of a photo-beam - whose occurrence is largely unpredictable.

In other cases, data acquisition could be triggered on a timebase. For example, in a testing lab a solenoid-driven piston jig is used to repeatedly press and release the 105 keys on a computer keyboard. Each key is connected to a circuit like that shown in Figure 5-2 (without the de-bouncing capacitor). The level of each line is read at 1 msec intervals for 250 msec following each press. In this way, the duration of switch

bounce in each key can be determined over a large number of cycles. This sequence is triggered by pulses produced by a 1-msec clock.

In theory, either event- or timebase-driven discrete digital I/O can be performed using any of the basic data acquisition modes described in Section 2.2.2 (polled, interrupt-driven, or DMA). In practice, event-driven digital I/O rarely uses DMA, since each event usually demands the attention of the program.

5.4.2 Triggering Discrete Digital I/O

Regardless of the source of the trigger event, the trigger signal - external event or clock pulse - must be connected appropriately to the digital I/O module. The options you have for making this connection depend on the mode of I/O and the functionality of the module.

For polled I/O, the trigger signal can be connected to any of the digital input lines or to the incoming handshake line used to signal "new data ready". If data input lines are used, the program can repeatedly read the input lines. When the trigger signal is detected, the program can take appropriate action. If handshake lines are used, the program can poll the appropriate status bit of the module's CSR. All digital I/O modules have a bit in the CSR which is set when the incoming handshake line is asserted. In the case of timebase-driven I/O, the clock pulse need not be connected to the digital I/O module at all, assuming the source of the clock pulses is a clock/counter module in the same system. The overflow bit of the clock's CSR can be polled directly.

For interrupt-driven I/O, the trigger signal must be connected to a line which is capable of generating interrupts when asserted. For many module types (for example, the DR11-C, the DR11-W, and the DRV11-WA), this implies that the trigger signal must be connected to the incoming handshake line. The DRV11-J allows an interrupt to be generated upon assertion of any of the 16 input data lines of port A, or any of the 4 incoming handshake lines (one for each of the 4 ports). Furthermore, the DRV11-J allows the user to select which state (TTL-high or TTL-low) is interpreted as "asserted" on a trigger line. As with polled I/O, an interrupt can also be generated directly by the clock if a system-resident clock/counter module is used for timebase triggering.

For timebase-driven DMA discrete digital I/O, the clock pulses must be connected to the incoming handshake line of the digital I/O module. Each clock pulse can then generate a DMA cycle request. Note that, though a "handshake" line is used for triggering, a full handshake is not

implemented. That is, the clock triggers a DMA transfer, but it does not complete the handshake by waiting for the digital I/O module to signal that the data have been successfully read. Thus, the user must insure that the triggering rate is within the performance limits of the module, or data will be lost. The same consideration applies to polled or interrupt-driven I/O triggered via handshake lines.

5.5 How To Perform Parallel Digital I/O

From a programming perspective, the most important characteristic of parallel digital I/O is the use of a handshaking protocol to synchronize the flow of information between the external device and the computer. In its simplest form, handshaking involves the exchange of two synchronizing signals each time a word (or byte) is transferred.

The sequence for exchange of a single data value is as follows:

1. The sending device puts a data value on the parallel link by placing a word or byte value in its output buffer register (or equivalent).
2. The sending device asserts a handshake line to signal that valid new data is ready.
3. The receiving device detects the "new data ready" signal and moves a word or byte from its input data buffer register into memory.
4. The receiving device asserts a second handshake line to signal that the data value was successfully read.
5. The sending device waits until it detects the return handshake signal before putting the next data value on the link.

Many devices, such as the DR11-C and the DRV11-J, automatically assert the handshake signals when the program reads or writes its data buffer. Thus, for example, when a DRV11-J is used as an output device, its "new data ready" line is asserted for approximately 400 nanoseconds when the program writes to the output buffer. No other explicit action on the part of the program is needed to assert the handshake signal. Likewise, the "data received" line is asserted whenever the input buffer is read in input mode.

In contrast, a non-DMA sending device must make provision for programmed detection of the "data received" signal from the receiving device. Similarly, a non-DMA receiving device must make provision for programmed detection of the "new data ready" signal from the sending device. This can be accomplished by polling status bits in the CSR or by enabling the module to generate an interrupt when these handshake lines are asserted. The status bits associated with these handshake signals are latched; that is, once the external device asserts the handshake line, the appropriate status bit in the CSR is set and remains set until cleared by the program. The status bit MUST be cleared as part of the read or write program sequence to insure proper operation of the handshake.

A common problem encountered when attempting to do parallel digital I/O is that the amplitude, duration, or polarity of the handshake signals do not match between the two devices. For this reason, you should determine these characteristics of the two devices before a final decision is made about what module to select. As part of this process, the programming logic for performing the handshake should be worked out in detail, based on the manufacturers' specifications for the two devices. If for some reason a match is not possible, external circuitry must be built to invert, prolong, or boost the handshake signals.

Chapter 6

How To Use The IEEE 488 Instrument Bus

The IEEE 488 bus is a standard method of interconnecting a variety of analytical instruments, computers, and computer peripherals. The bus allows computers to control instruments and pass data between them.

The IEEE 488 is particularly useful where a variety of medium- to high-speed instruments from different vendors are involved. As a computer-to-instrument link, the IEEE 488 is faster and easier than RS232 when more than a few instruments are involved. It is far superior to analog interfacing in terms of noise immunity, convenience, speed, and reliablility.

A typical IEEE 488 system consists of a computer with IEEE 488 interface option, instruments with built-in IEEE 488 connectors, and IEEE 488 cables to connect the computer to the instruments in a daisy chain.

The cable, the connectors and the electrical characteristics of the cable are specified by the IEEE standard. The software to control IEEE 488 instruments is not yet covered by a standard.

Up to 15 devices may be configured on one bus. Devices include not only instruments and computers, but also computer peripherals such as printers and plotters. The length of the bus is restricted to 15 meters or 2 meters per device. Data rates across the bus are limited by the slowest device. Typically, the maximum data rates are between 100,000 to 200,000 bytes (digits) per second.

Most IEEE 488 computer interfaces, such as the VAXlab IEQ11-AD, are available with both driver and subroutine software. This software simplifies the programming of IEEE 488 instruments.

The IEEE 488 cable consists of 16 wires, which are called *bus lines*. These lines are shared by all instruments and computers on the bus. Eight of the lines, called *data lines*, are used to encode the messages sent on the bus. Three more lines, called the *handshake lines*, are used to make certain that each character sent is accurately received. The remaining five lines are for *general bus management*.

6.1 Talker, Listener, Controller

Instruments play well-defined roles on the bus. An instrument sending a message is called a *talker*. Only one instrument may talk at any one time. An instrument receiving a message is called a *listener*. Any number of instruments can listen to the message being sent by the talker. Instruments can be talkers only, listeners only, or talkers and listeners. The user's guides for your instruments will tell you what your instruments are.

The computer is called the *controller* of the bus. As such, it tells bus instruments when to talk and when to listen. No instrument can ever talk or listen unless told to do so by the computer. The computer controls all bus activity, and it must be the only controller of the bus. This means that no other device, not even another computer, can be a controller on this IEEE bus. The computer can make itself a talker or listener, however, it listens to all traffic on the bus.

For example, suppose your IEEE bus system consists of a computer, a multimeter, and a signal generator. You want the computer to receive from the multimeter a message that reports a voltage reading, and then send the signal generator a message that causes it to generate a signal based on the voltage reading. To receive the voltage reading, the computer tells the multimeter to be the talker, and the computer itself is the listener. The computer tells the signal generator to neither talk nor listen, so the signal generator ignores the message that the multimeter sends to the computer (see Figure 6-1). To send instructions for the signal output, the computer tells the signal generator to be a listener, and the computer itself is the talker. The computer tells the multimeter to neither talk nor listen, so the multimeter ignores the message that the computer sends to the signal generator (see Figure 6-2).

Figure 6-1: Computer Receives a Message

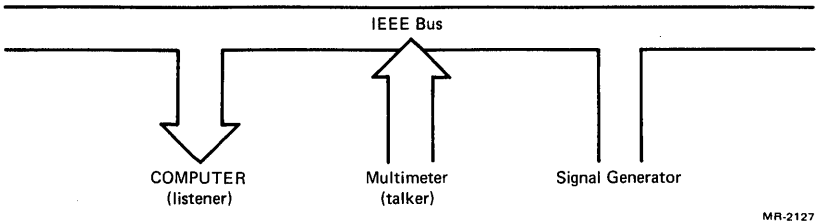
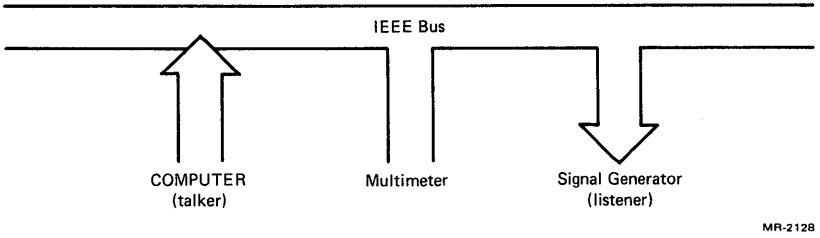


Figure 6-2: Computer Sends a Message



6.1.1 Instrument Addresses

Each instrument on the bus has a number between 0 and 30 that the computer uses to identify the instrument when the computer tells the instrument to either talk or listen. This number is the instrument's address and can be set with switches located on the instrument itself. Before you can use the IEEE routines, you must know the addresses of the instruments on your IEEE bus. In the previous example, you might set the multimeter to have an address of 1, and the signal generator to have an address of 2. An instrument's address is also called its *primary address*.

Some bus instruments have different functions or parts that the computer can specify by using a *secondary address* in addition to the instrument's primary address. Secondary addresses are in the range 0 to 30, though in order to distinguish them from primary addresses, they are specified in IEEE bus routines by numbers in the range 200 to 230. Each instrument's designer defines the meanings of any secondary addresses the instrument recognizes. For example, when you tell the multimeter above to talk, it might report a voltage reading if you specify secondary address 3, and a resistance reading if you specify secondary address 1.

We suggest that you write down the address of each bus instrument, along with any secondary addresses and what they specify, on a form such as the one shown in Figure 6-3. Be sure that no two instruments have the same primary IEEE bus address.

Figure 6-3: Sample Address Record

IEEE Bus Addresses

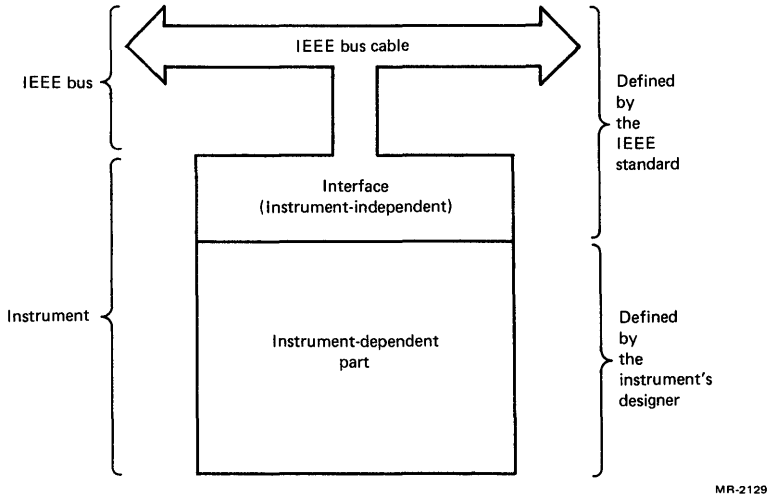
Instrument	Address	Secondary Addresses
Multimeter	1	1 resistance
		2 amperage
		3 voltage
Signal Generator	2	none

MR-2126

6.1.2 Interface

Part of each instrument on the IEEE bus is defined by the IEEE standard and thus is not instrument-dependent. This part is called the instrument's interface to the bus. The rest of the instrument is not defined by the standard, but by the instrument's designer. These parts are illustrated in Figure 6-4. Because every instrument on the IEEE bus has an interface, the bus is sometimes called the *interface bus*.

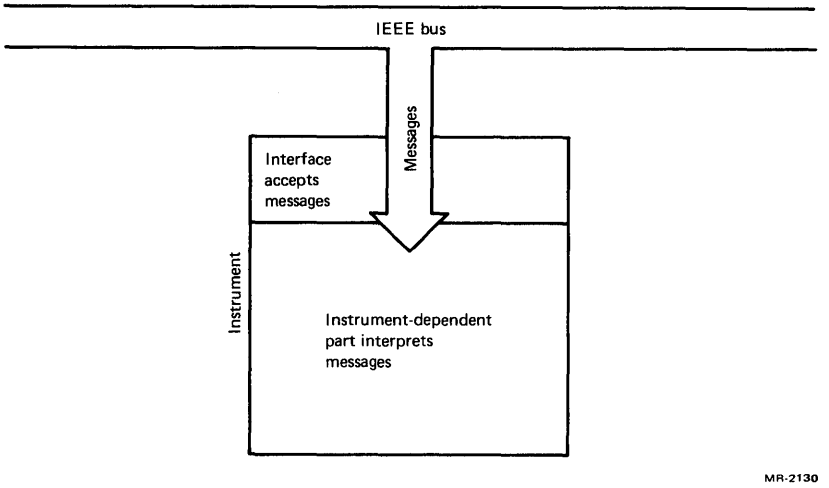
Figure 6-4: An Instrument's Interface



Only an instrument's interface interacts directly with the bus. Messages are interpreted by the instrument-dependent part of the instrument, but they are sent and received through the interface. When the computer tells the instrument to listen, the instrument's interface passes any subsequent messages sent on the bus to the instrument-dependent part of the instrument. This is illustrated in Figure 6-5.

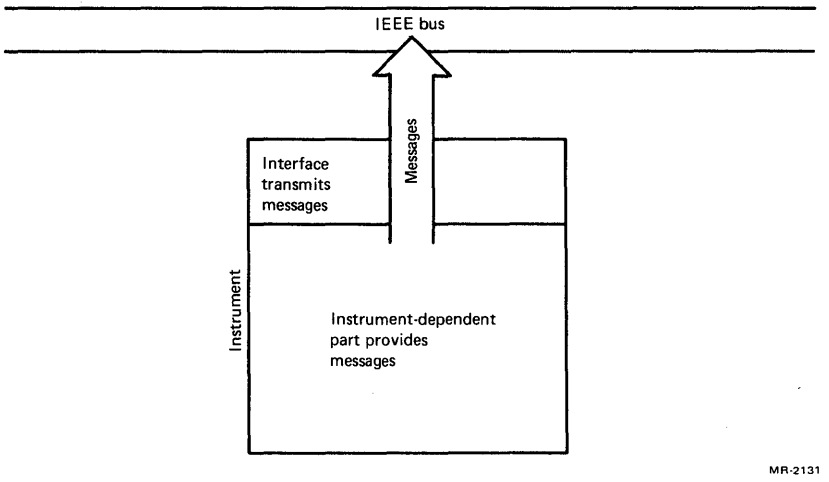
When the computer tells the instrument to talk, the instrument's interface transmits messages from the instrument-dependent part of the instrument. This is illustrated in Figure 6-6.

Figure 6-5: An Instrument Listens



The computer controls the bus by sending commands, which are instructions to the interfaces on the bus. This is illustrated in Figure 6-7. Like messages, commands are sent as characters on the data lines; but, unlike messages, commands are intercepted and interpreted by the interface, not passed to the instrument-dependent part of the instrument. The interface interprets each command according to the meaning defined for that command by the IEEE standard. Only the computer can send commands. The computer uses commands to tell instruments to talk or listen.

Figure 6-6: An Instrument Talks



6.2 How To Communicate With An Instrument

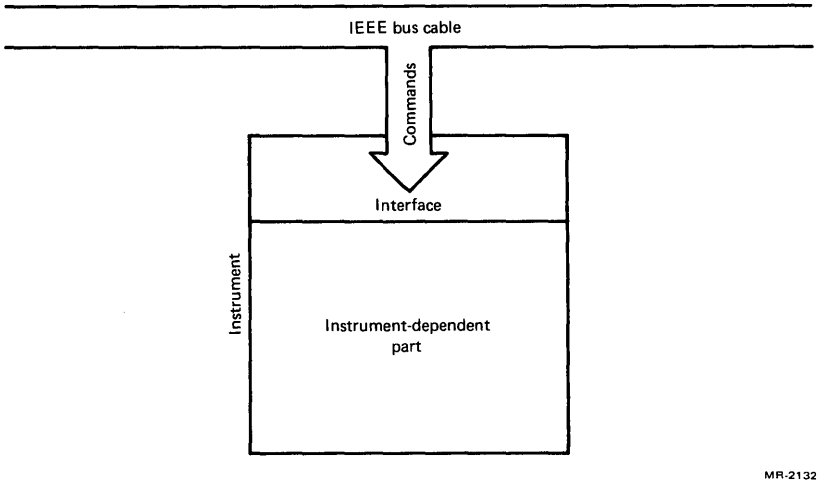
6.2.1 Messages

The contents of each message string and the effect it has on the information it reports are as varied as the types of instruments on the bus. The IEEE standard only defines how instruments communicate, not what they communicate. For this reason, the user's guide for each instrument is an essential source of informaton when you use the IEEE routines. It helps you decide what strings to send to that instrument and tells you what strings you should expect to receive from it.

For example, the multimeter above might send back a reading with these characters:

V+4.382E+01

Figure 6-7: An Instrument Accepts Commands from the Computer



where the "V" indicates that this was a voltage reading (not an amperage or resistance reading), and the other characters indicate a measurement of 43.82 volts. Based on this reading, your program might tell the signal generator to generate a 43.8 volt at 1250 Hz. The message telling the signal generator to do this might be:

V43.8F1250

where "V43.8" means "43.8 volts" and "F1250" means "at a frequency of 1250 Hz."

6.2.2 Sending Messages

When the computer itself is the talker, it sends a message string specified by your program to the listeners specified by your program. Not all instruments are able to listen.

Each instrument that can listen has a vocabulary of characters and a syntax that are meaningful to it. Because vocabularies differ from instrument to instrument, you usually send each message to only one instrument. The user's guide for a particular instrument lists the characters the instrument recognizes and the effect of each character. The message you send to an instrument depends on the effect you want and which character or characters cause that effect.

One of the five general management bus lines is known as the *End of Identify* (EOI) line. The talker sets the EOI line while sending the last character of its current message, thus indicating the end of that message to the listeners. Some instruments do not act on any message characters until the EOI line is set. The EOI line could also be set by common terminating characters.

6.2.3 Receiving Messages

When the computer is a listener, it stores the message string sent by the talker you specify. Not all instruments are able to talk. For example, the signal generator might be able only to listen.

The meaning and format of the string is determined by the instrument's designer. The user's guide for an instrument tells you what information the instrument sends and the format of its messages.

The computer knows the message is complete when one of the following three conditions occurs:

- The talker sets the EOI bus line while sending a character. For example, the multimeter above might set the EOI line while it sends the character "1".
- The talker sends a terminator, a character the computer recognizes as an end-of-message indicator. There are two types of terminators, those recognized by the computer and those recognized by the instruments. Normally, the computer recognizes a carriage return or a line feed as a terminator, but you can often change which characters are terminators. For example, the multimeter might follow the "1"

character with a carriage return character. Note that if it did this, it could not set the EOI line while sending the "1". Depending on how it was designed, an instrument might or might not set the EOI line while sending the carriage return character. Your instrument manual will tell you what character your instrument recognizes as a terminator.

- The number of characters sent in the current message reaches a limit set by your program. For example, if the multimeter is not designed to set the EOI line or send a terminator, your program should specify the number of characters in the string as the maximum number of characters in the message.

Note

If the instrument sends more characters than the limit set by the program, the remaining characters are not lost by the system. Any further transmissions by the talker are delayed until the program takes some action to retrieve the remaining characters.

6.3 How To Test The Status Of An Instrument

Any instrument on the IEEE bus can report information about its current status to the computer. Though the type of status information reported depends on the particular instrument, the procedures to report status are part of the IEEE standard, and are the same for all instruments.

As controller of the IEEE bus, the computer can ask instruments for their status by conducting either a serial poll or a parallel poll. In a serial poll, instruments report status information, one at a time, on the data lines of the bus. Since the bus has eight data lines, each instrument can report up to eight bits of status information. In a parallel poll, up to eight instruments can simultaneously report status information on the data lines. Each instrument can use only one data line, and can thus report only one bit of status information. The bit of information reported in a parallel poll is not necessarily related to any of the information reported in a serial poll. A particular instrument could respond to one, both, or neither of these types of polls. The user's guide for a particular instrument tells you which polls that instrument can respond to, and what status information it reports.

There is one type of status information an instrument can tell the computer without having to wait to be polled. It can tell the computer that it needs service. It issues this service request by setting a bus line reserved for this purpose, the SRQ (service request) bus line. If your program ignores the service request, the instrument can take no further initiative. Setting the SRQ line is the only action on the IEEE bus that instruments can initiate independently of the computer.

Since bus lines are shared by all instruments, the computer doesn't know which instrument is setting the SRQ line. One function of a serial poll is to give the controller a way to find out which instrument is requesting service. As part of the informaton in its serial poll response, each instrument must report whether or not it is requesting service. Any instrument that can request service must be able to respond to serial polls.

6.3.1 Serial Polls

If more than one instrument is to be polled, the computer polls the instruments one after the other.

Each instrument polled has eight bits of status information in its interface; this group of bits is called the instrument's status byte. Each bit of the status byte is set (1) or clear (0) to report specific information about the instrument. When the computer serially polls the instrument, the instrument reports the state of each of these bits by setting or clearing the corresponding data line on the bus.

As shown in Figure 6-8, data line 6 contains the same type of information for all instruments. An instrument sets this line in response to a serial poll if it is requesting service from the computer. This line is different from the SRQ line, which is not one of the data lines. If an instrument has set the SRQ line, however, it must also set data line 6 when it is serially polled. Each instrument's designer determines the type of informaton that the instrument reports on each of the other data lines.

6.3.2 Service Requests

Your program can detect a service request in either of two different ways. Which you choose will depend on how you wish to perform the detection operation itself and how you wish subsequent serial polls to be initiated.

Figure 6-8: Serial Poll Response

Data Line	Meaning
7	Meaning is instrument-dependent
6	Set if the instrument is setting the SRQ line
5	Meaning is instrument-dependent
4	Meaning is instrument-dependent
3	Meaning is instrument-dependent
2	Meaning is instrument-dependent
1	Meaning is instrument-dependent
0	Meaning is instrument-dependent

MR-2123

One way to detect a service request is to designate a service subroutine. In this case, the IEEE 488 device driver assumes the resonsibility of detecting a subsequent service request and initiating a serial poll. This procedure is identified as a "driver-initiated" serial poll.

A second way to detect a service request is to test the state of the SRQ line and, when the line is found to be asserted, to perform a serial poll. This allows your program to retain the responsibility of detecting service requests. This procedure is identified as a "user-initiated" serial poll.

Driver-initiated handling of service requests begins with detection of the SRQ-asserted condition. Once this occurs, the driver serially polls the list of devices capable of generating service requests. When a device is found to be requesting service, the service subroutine is called by the driver. This subroutine is user-written and has as its purpose the servicing of instruments requiring sevice.

When a service request is detected, your program receives the instrument address of the device which asserted the SRQ line. The variable which receives this information is used by your service subroutine to determine which instrument requested service and therefore the type of service to be rendered.

The action required when an instrument requests service depends on the characteristics of the particular instrument. For example, one instrument on your IEEE bus might request service when it has new data; your program would then ask it for the data. Another instrument might request service when it is out of paper; your program would type a warning on the terminal.

6.3.3 Parallel Polls

Parallel poll capability allows instruments to quickly notify the controller of their status. Every instrument that can do so responds to the poll. Each instrument polled has in its interface a bit of status information called the *status bit*. The meaning of each instrument's status bit is determined by the manufacturer and is described in the user's guide for the instrument.

An instrument can respond to a parallel poll only if it is designed to respond and if its response has been enabled. Any instrument that can respond to a parallel poll must be able to have its response enabled either by local controls or by the computer, but not by both. The method of local poll enabling is not part of the IEEE standard, but is determined by the instrument's designer.

An instrument can respond to parallel polls until its parallel poll response is disabled. If the response was enabled locally, it is also disabled locally.

Enabling an instrument's parallel poll response assigns the instrument one of the bus's data lines and a condition. The instrument sets the data line during a parallel poll if the status bit in its interface is in the assigned condition. If the condition assigned to it is 0, the instrument sets the data line if its status bit is 0 at the time of poll. If the condition assigned to it is 1, the instrument sets the data line if its status bit is 1 at the time of poll.

For example, suppose your program enables instrument 17 with data line 5 and condition 0, and enables instrument 15 with data line 2 and conditon 1. When your program conducts a parallel poll, the data lines have the values shown in Figure 6-9.

Figure 6-9: Example of a Parallel Poll Response

Data Line	Value
7	0
6	0
5	0 if instrument 17's status bit is 1 1 if instrument 17's status bit is 0
4	0
3	0
2	0 if instrument 15's status bit is 0 1 if instrument 15's status bit is 1
1	0
0	0

MR-2124

Data Line	Associated Value
0	1
1	2
2	4
3	8
4	16
5	32
6	64
7	128

MR-2125

More than one instrument can be assigned to a single data line. However, this is not usually done, because your program could not detemine which instrument set the line, if it were set during a parallel poll. Each of the eight data lines is clear in a parallel poll unless one or more instruments sets it.

Note

When more than eight instruments must be polled, two or more instruments may be assigned the same data line for their poll reply signal. If this is done, a logical one on the data line would signify that at least one of the associated instruments had replied to the poll. A logical zero would signify that none of those instruments had replied.

6.4 Remote And Local States

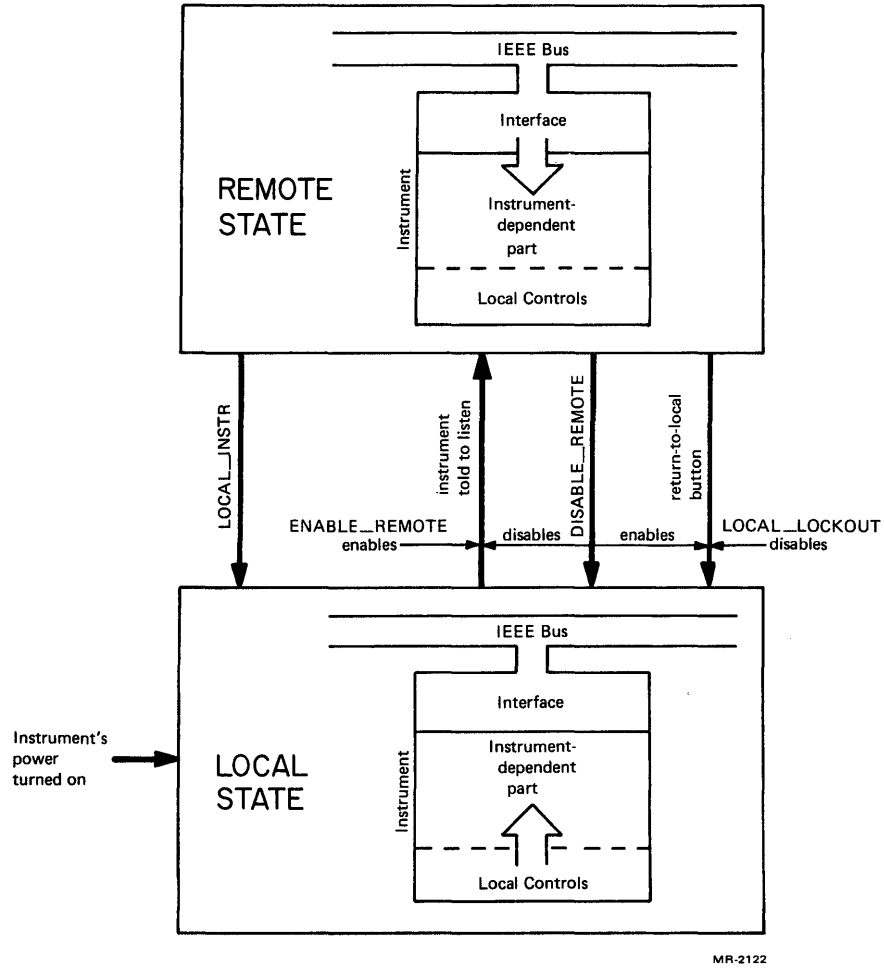
An instrument on the IEEE bus can use input information either from the bus or from manual controls on the instrument itself. When messages from the IEEE bus are the source of input information, the instrument is said to be in the *remote state*. When the manual controls on the instrument are the source of input information, the instrument is said to be in the *local state*. This section discusses how your program can control which state each instrument is in. These controls are summarized in Figure 6-10. A heavy arrow in this figure represents a transition between the local and remote state. A light arrow represents a statement that enables or disables the transition that the arrow points to.

No instrument can be in the remote state unless the remote enable (REN) bus line is set. When you start the computer, this line is clear, so all instruments on the bus are in the local state. The first IEEE bus routine to execute after the computer is started sets the REN line automatically. This line remains set while the computer communicates with instruments on the bus. An instrument enters the remote state when the computer tells it to listen while the REN bus line is set.

Some instruments have a return-to-local button, which can be used to put the instrument in the local state. The user's guide for a particular instrument tells you whether or not that instrument has a return-to-local button and, if so, where it is.

The instrument designer determines how the instrument behaves under local control and under remote control. When going from local to remote control, an instrument can either use its current local settings until they are overridden by remote input or use remote input that was previously received. In either case, the instrument must ignore future use of its local controls and become responsive to remote input. Some instruments, however, have functions that are always controlled locally, even in the remote state. When going from remote to local control, an instrument can either use input from its local controls immediately, or continue to use the last input from the bus until that input is overridden by subsequent local control settings. In either case, the instrument must ignore future remote input and respond to future use of its local controls. It can still talk and listen while in the local state.

Figure 6-10: Remote and Local States



6.5 How To Reset An Instrument

Your program can separately clear, or reset, either the instrument's interface to the bus or its instrument-dependent part.

6.5.1 Clearing Interfaces

The bus and every instrument's interface to the bus can be cleared by setting the *interface clear* bus line (IFC), a line reserved for this purpose by the standard. Setting this line clears only the interfaces, not the instrument-dependent parts of instruments. Each interface returns to the clear state defined by the IEEE standard. Setting the IFC line has the following effects:

- All return-to-local buttons of bus instruments become operative.
- The REN bus line is cleared and then set.
- All instruments enter the local state. However, because the REN bus line is set, each instrument enters the remote state when it is told to listen.
- Any condition set by a message is not affected by this routine.

You can use an interface clear command at the beginning of your program to undo any effect that previous IEEE bus routines have had on the interfaces.

6.5.2 Clearing Instruments

Instrument clear routines clear the instrument-dependent parts of instruments on the bus. These routines do not clear the instrument's interfaces. Each instrument cleared returns to a clear state defined by that instrument's manufacturer; this is usually the state the instrument is in after its power is turned on. Refer to the user's guide for the particular instrument for the properties of this state.

You can use these routines at the beginning of your program to undo the effects of prevous message routines.

CAUTION

Never turn on an instrument's power while the computer is running!

Chapter 7
How To Control Serial Devices

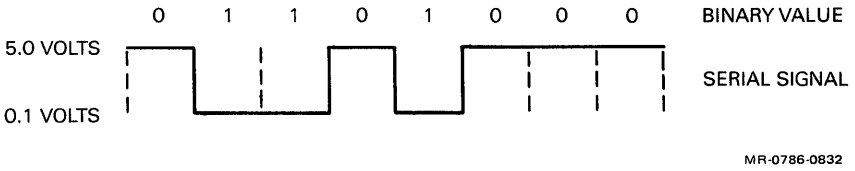
In the modern laboratory, many instruments have built-in "intelligent" controllers. These controllers are microprocessors which do much of the work that might otherwise have to be done by a general-purpose computer, such as digitization of analog signals, control of solenoids using discrete digital signals, setting control voltage levels, and various forms of data reduction. For example, many chromatographs and spectrometers have integral microprocessors which digitize the detector voltages, analyze the results, and output peak parameters via a serial data port either to a printer or to a host computer.

In order to interface such instruments to either printers or lab computers, it may be necessary to build customized cables, reconfigure the serial ports, or write specialized programs to handle the data interchange. Before presenting a detailed discussion of each of these topics, an overview of serial data transmission is given.

7.1 Serial Data Communication

The term *serial data communication* refers generally to any scheme in which digital data is transferred bit-by-bit. For example, if an eight-bit byte were transferred serially, the bits would be sent one after the other, in sequence. Since each bit could be either a 1 or a 0, the state of each bit is represented as a voltage level, much as in parallel digital data communication. Thus, the byte 01101000 (binary) could be represented by the following sequence of voltage levels: 5, .1, .1, 5, .1, 5, 5, 5, where 5 volts represented a binary 0 and .1 volts represented a binary 1. This is shown diagramatically in Figure 7-1.

Figure 7-1: A Simple Serial Data Stream



This seems fairly simple, until one considers several important questions:

- What are the voltages used to represent binary 1s and 0s?
- How long is each bit's representative voltage held at a steady level; that is, how rapidly does the pattern change?
- How is the beginning of a byte pattern signalled?
- How does the transmitting device know whether or not the receiving device is ready?
- How are messages - strings of bytes - delimited?

In order for two devices to communicate via a serial data link, some agreement regarding these and other details is necessary. Many of the elements of a standardized protocol for serial data communication are contained in the EIA RS-232 standard. Though the RS-232 standard defines many of the parameters of the protocol very rigidly, some may vary from one implementation to another, depending on the requirements of the application. Furthermore, many manufacturers adhere very loosely to the standard, inventing seemingly endless variations. For these reasons, it is often quite difficult to get two "standard" RS-232 devices to communicate with each other.

One reason for this confusion is that the RS-232 standard was developed before the advent of the microprocessor. Intelligent instruments, microcomputers, and, for that matter, serial printers were unknown in their present forms. The standard was developed to regulate communication between teletype terminals over telephone lines via modems. Thus, many of the concepts implicit in the RS-232 standard are based on modem/phone line properties rather than on the properties of modern intelligent instruments and peripherals.

The sections below outline the major issues one must address to successfully establish a serial data communication link between two RS-232 devices, be they intelligent instruments, computers, printers, or CRT terminals.

7.2 How To Configure Devices For RS-232 Compatibility

Serial communication is performed by the computer using an I/O option module which is, in many ways, like an analog or parallel digital module. Such modules are sometimes referred to as *serial line units*, or SLUs. Like other types of modules, they are usually configurable using jumpers or switch packs. More sophisticated serial option modules are software configurable - that is, the most important parameters can be set by writing appropriate values into the module's registers. Similarly, the serial port of an intelligent laboratory instrument or a printer is frequently configurable, rather than having set characteristics. The following characteristics of the serial port are commonly settable:

- The baud rate
- The number of start and/or stop bits

- The number of data bits
- The type of parity used
- The type of handshaking used

7.2.1 Selecting A Baud Rate

The term *baud rate* refers to the rate at which data is transmitted, expressed in bits-per-second. Within any given byte pattern, the timing of pulses representing each binary digit is synchronous; that is, each bit-pulse occurs at a set time following the onset of the byte pattern. Since nothing distinguishes one bit from another, other than their timing, it is important that both sender and receiver agree on the baud rate.

Common rates used in the U.S. are 300, 1200, 2400, 4800, and 9600 baud. Obviously, the baud rate affects the overall rate of data transfer. Since a byte is 8 bits, one might presume that the data rate in bytes-per-second might be computed as the baud rate/8. However, since all RS-232 protocols include at least one start bit and one stop bit, the proper rule-of-thumb formula for computing the theoretical maximum data rate is baud rate/10. This rate frequently is not achievable in practice (see Section 7.2.5). Ordinarily, though, optimal throughput is attained by configuring both devices for the maximum baud rate of which BOTH are capable.

7.2.2 Selecting The Numbers Of Start And Stop Bits

Under the RS-232 standard, byte patterns always begin with a low-level (1) pulse and end with a high-level (0) pulse. These are called the *start* and *stop* bits, respectively. Thus, each byte always begins with a high-to-low transition - that is, the transition between the stop bit of the preceding byte and the start bit of the current byte. These bits have no information content; they are used only as synchronizing signals.

Though start and stop bits are always present, their durations can vary from one device to another. The durations of the start and stop bits are expressed in terms of the number of each, since the duration of a single bit is determined by the baud rate. Furthermore, the numbers of start and stop bits can be nonintegral. For example, many devices use 1.5 stop bits. This implies that following the last data bit (or parity bit, if present), the signal is guaranteed to be high for at least 1.5 times the duration determined by the baud rate.

The receiving device interprets the signal level as representing successive binary digits beginning NS/baud seconds following the onset of the first start bit, where NS is the agreed-upon number of start bits. Thus, both devices must agree on the number of start bits.

However, stop bits are a different story. The receiving device interprets any low signal level between the end of the last data (or parity) bit and the end of the last stop bit as an error in transmission, called a *framing error*. If the receiving device were set for one stop bit and the sending device were set for 1.5 or 2 stop bits, the protocol would work properly. The receiving device would assume only that the sending device was a little slow, as might be the case when it was too busy to send each byte immediately following the last.

For maximum efficiency, configure both devices for the minimum number of start and stop bits that BOTH devices will accept. If it is not possible to match the number of stop bits, the next best thing is to configure the receiving device for fewer stop bits than the sending device. (Note that this assumes that data flows in one direction only. Note also that XON and XOFF characters may be sent from the receiver to the sender even when there is only one-way data transmission.)

7.2.3 Selecting The Number Of Data Bits

The RS-232 standard applies to serial data transmission in bit-synchronous bytes. Ordinarily, a byte is considered to be 8 bits wide. However, if the data are exclusively ASCII characters, only 7 bits are used; the top bit is always 0. Since it takes less time to send 7 data bits per byte than 8, many devices either require or allow that only 7 bits be transmitted.

If either device is exclusively an ASCII device, configure both for 7-bit transmission, if possible. If the data being transmitted is binary, that is, if the data stream contains byte values in the range 127-255 (decimal), set both devices for 8-bit transmission. Both devices MUST be similarly configured.

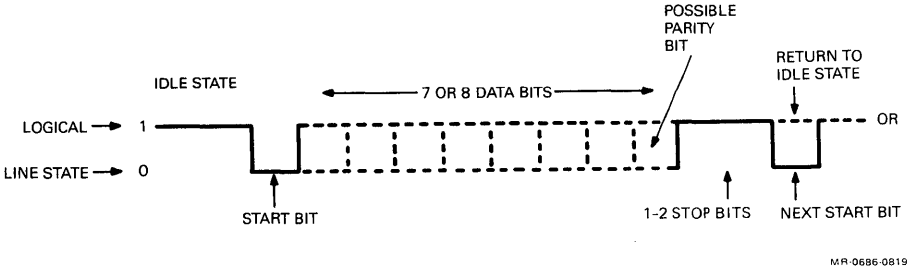
7.2.4 Selecting The Type Of Parity Used

In electrically noisy environments, false positive or negative signals can interfere with the accurate transmission of data over a serial line. To insure against this, many devices implement a strategy for testing each byte of data to verify that it has been properly received. An extra bit, called a *parity bit*, is added to each byte pattern, between the last data bit and the stop bit. The sending device counts the number of 1s in each byte and then appends a parity bit whose value is either 1 or 0, so that the total number of 1 bits (excluding the start and stop bits) is either an odd number (*odd parity*) or an even number (*even parity*). The receiving device counts the number of 1 bits in each byte and signals an error condition if the sum does not match the parity (even or odd).

Many devices are configurable for even parity, odd parity, or no parity (parity disabled). Since the parity bit is one more bit added to each byte, the overall speed of transmission is reduced when parity is used. In environments in which noise is not a problem, disable parity, if possible. If the transmission line is electrically noisy, as is the case with modems, for example, select odd or even parity. Either will do as long as it is the same on both ends. It may also be advisable to determine whether the hardware and/or software can correct parity errors by forcing retransmission. If not, enabling parity only allows you to detect errors in transmission, not correct them.

At this point, all the major variations in the format of serial data have been covered. Figure 7-2 illustrates that format diagramatically and describes its possible variations.

Figure 7-2: Serial Data Protocol



7.2.5 Selecting A Method Of Handshaking

By setting the baud rate, parity, and number of start and stop bits, you can insure that the sending and receiving devices are synchronized with respect to the transmission of each data byte. However, you may still need to synchronize the two devices at the byte-stream level. Suppose, for example, that a computer was trying to send control information to an intelligent instrument through a 9600-baud serial port. Theoretically, data could flow at a rate of about 1000 bytes/sec. But what if the instrument's processor needed to attend to some other task while information was being sent? This could occur if the information being sent was a series of instructions, each of which was executed by the instrument before the next was accepted. How could you be sure that bytes of information were not lost while the instrument's processor was inattentive to the serial port?

One way, of course, would be for the serial port device to buffer incoming data in its own internal memory. Such memory buffers are found in many serial port devices. Commonly, the buffer is set up as a FIFO (first-in/first-out buffer), into which data bytes are placed as they arrive and from which the processor removes bytes when it is ready to read from the serial port. However, serial FIFOs, when they are implemented at all, often have very limited storage capacity. Four bytes is a common FIFO capacity. Whatever the capacity of the FIFO, it only provides a finite margin of safety. Once it is filled, the processor must remove a

byte before another can be received. If a byte is transmitted while the FIFO is full, data is lost, and the SLU signals an error condition by setting a bit in its CSR.

Whether or not the serial device has a FIFO, it is desirable for the receiving device to be able to inform the sending device that it (the receiver) is unable to accept more data. Methods for accomplishing this are called *handshaking* protocols. A handshake is basically a message that goes between the devices that says "I can't take more data now." or "OK. I'm able to take more data now." There are two ways of performing handshaking in serial data communication:

- By using a "hardware" signal whose level indicates the readiness of the receiver
- By using "software" signals - special ASCII characters transmitted over the data lines

7.2.5.1 Hardware Handshaking

Hardware handshaking in serial data transmission is implemented using dedicated signal lines in addition to the lines on which the serial data are transmitted. When a device is ready and able to accept data, it asserts the line which indicates "I am ready" to the other device. The sending device checks this signal before transmitting each data byte. The sender will wait until this signal is asserted before sending new data. Once the transmission of a data byte is begun, transmission completes even if the receiver deasserts the "I am ready" signal during transmission. Thus, the receiving device ordinarily deasserts this signal whenever its FIFO is filled to within one byte of capacity (or less, in some cases). As soon as more space becomes available in the FIFO (that is, when data are transferred out of the FIFO to main memory), the receiving device reasserts its "I am ready" signal and the sender resumes transmission.

7.2.5.2 Software Handshaking

Software handshaking is conceptually like hardware handshaking; but, instead of using dedicated signal lines for the "I am ready" signals, the data lines are used. When the receiving device's FIFO is filled to within one byte of capacity (or less), the receiver sends a single character to the sender - the ASCII character DC3 (hex 13, also called XOFF). When more room becomes available in its FIFO, the receiver sends the ASCII character DC1 (hex 11, also called XON) to the sending device to indicate that transmission can be resumed.

Whereas the hardware handshaking protocol is handled entirely by the SLU, transparently to the computer, software handshaking must be handled by the programs which control the serial ports on both devices. In the case of an intelligent instrument or peripheral device, software handshaking is performed by the firmware of the device. In the case of a computer system, it is usually done in the device driver for the SLU. When connecting two serial devices, you must determine what form of handshaking both devices can use (hardware or software). Then, set whatever jumpers or switches are necessary on both the SLU and the external device to select the type of handshaking desired. Given a choice, you should select hardware handshaking, since it involves less overhead on the part of the CPU.

7.2.5.3 Devices Without Any Handshaking

Some serial devices do not use any form of handshaking. This situation is most often found in "send only" devices like digitizing tablets. In these cases, the serial link becomes a true realtime data stream, in that the receiving device must be able to process received characters at some minimal speed, or data will be lost. To insure that no data is lost, the speed of transmission should be set to a low baud rate. Also, the speed of transmission can be lowered by increasing the number of start, stop, parity, and data bits in each byte, as described in the preceding sections.

7.2.5.4 ACK/NAK Protocol

The hardware and software handshaking techniques described above can be used to insure that the sending and receiving devices are synchronized at the byte-stream level. Some intelligent instruments implement an even higher-order protocol for insuring that any errors in data transmission (due to noisy lines, for example) are detected and corrected. Briefly, this is accomplished as follows.

1. The sending device computes a checksum which it appends to the end of each message. In the present context, a message is a string of bytes constituting a unit of information, often a line of ASCII characters delimited by a carriage return.
2. The receiving device computes the checksum based on the data alone, and compares what it computes with the checksum from the sender.
3. If the two checksums match, indicating valid transmission of the message, the receiver sends an ASCII ACK (acknowledge) character to the sender. If the two checksums do not agree, the receiver sends back a NAK (negative acknowledge) character.
4. The sender waits for either an ACK or a NAK character in response to each message transmitted. If the sender gets an ACK, it sends the next message. If the sender gets a NAK, it retransmits the preceding message and continues to do so until the receiver returns an ACK.

7.3 How To Connect Serial Devices

In order for two devices to communicate via an RS-232 serial link, the data and handshaking lines must be properly connected. The RS-232 standard defines names, functions, and even connector pin numbers for many different signal and handshaking lines. The discussion of connections begins with a description of a terminal/modem linkage since the functions of the various lines are much more easily understood in their original context. In subsequent sections, the implementation of the RS-232 standard for computer/instrument linkages is discussed.

7.3.1 Terminal/Modem Connection

The full RS-232 standard defines many handshake and status lines, as well as both primary and secondary data paths. In practice, only a subset of these lines is used in most implementations. For present purposes, only two data lines and six handshake lines need be described.

Each line, whether it carries data or handshaking signals, is controlled by one device (terminal or modem) and "read" by the other device. That is to say, all lines are unidirectional. The cable connecting the terminal with the modem is a "straight-through" cable with a 25-pin "D" connector at each end. For example, pin 2 of each connector is defined as the pin on which data originating in the terminal is carried. This arrangement implies that pin 2 on the bulkhead of the terminal is wired to a transmitting circuit within the terminal, while pin 2 on the bulkhead of the modem is wired to a receiving circuit within the modem. This same principle applies to all data and handshaking lines. Thus, the circuit connections of the bulkhead connector on the terminal are different from the circuit connections of the bulkhead connector on the modem. To discriminate between the two, the bulkhead connector of the terminal is referred to as a *data terminal equipment*, or DTE, connector and the bulkhead connector of the modem is referred to as a *data communication equipment*, or DCE, connector.

Figure 7-3 shows how DTE and DCE connectors are wired together. The names and directions of each signal are indicated.

Protective Ground is the chassis ground of the two devices and is also connected to the cable shield.

Transmitted Data carries data originating in the terminal to the modem.

Received Data carries data originating in the remote device passed via the modem to the terminal.

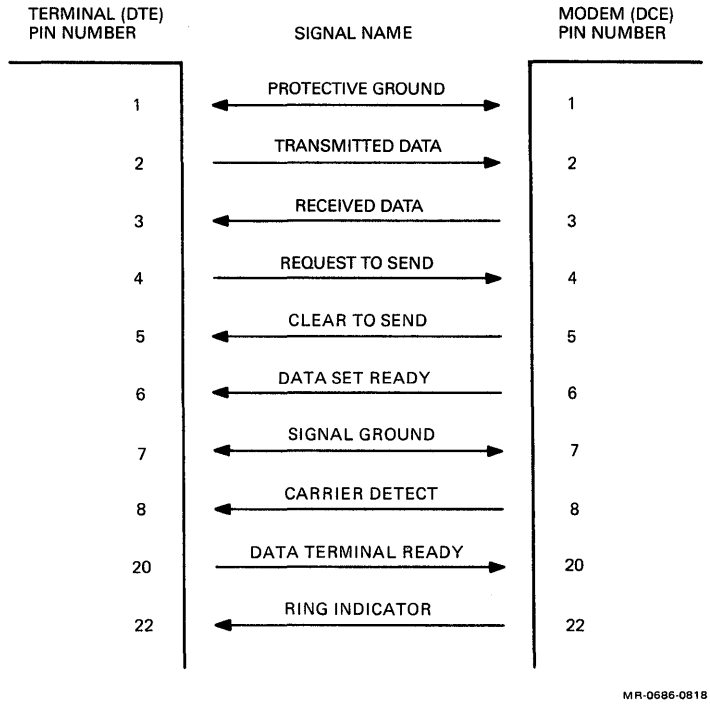
Request to Send (RTS) is asserted by the terminal when it is ready to transmit a data byte.

Clear to Send (CTS) is asserted by the modem when it is ready to pass a data byte on to the remote device.

Data Set Ready (DSR) is asserted by the modem when it is powered up and in an active state (as opposed to a test state).

Signal Ground (GND) is the circuit ground for all data and handshake lines.

Figure 7-3: DTE and DCE Signal Connections



Carrier Detect (CD) is asserted by the modem when it is receiving a carrier signal from the remote modem. That is, CD indicates that the local modem has a live connection to its remote counterpart. This line is also called the *Received Line Signal Detector* (LSD).

Data Terminal Ready (DTR) is asserted by the terminal when it is powered up and in an active state (as opposed to a test state).

Ring Indicator (RI) is asserted by the modem when a ring signal is being received on the telephone link to the remote modem.

When a key is pressed on the terminal, a data byte becomes available to be transmitted to the remote device via the modem. First, the terminal asserts RTS. If CTS, DSR, and CD are all asserted, the terminal sends the data byte to the modem. It then deasserts RTS and pauses until another key is pressed.

When a character is received by the modem from the remote device, the modem sends the character immediately to the terminal, if DTR is asserted.

7.3.2 Computer/Device Connection

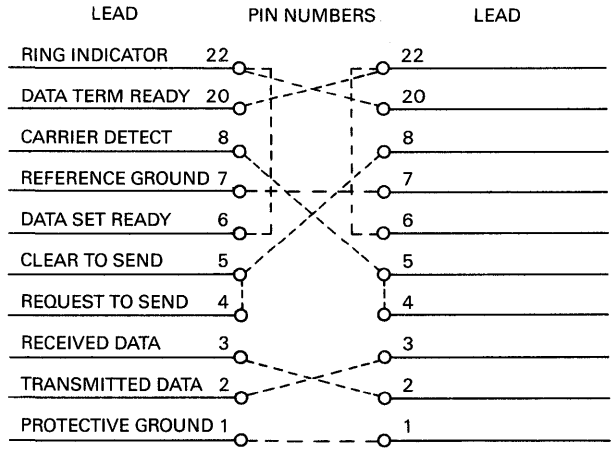
When two RS-232 devices, such as an SLU module and an intelligent lab instrument, are connected directly to each other, some subset of the RS-232 lines is usually implemented. Which lines are used and how their function is defined depends on the design of the device's serial port interface and the design of the SLU. If no hardware handshaking is implemented, the minimal number of lines that must be connected for bidirectional communication are Transmitted Data, Received Data, and Signal Ground. If hardware handshaking is used, the handshake lines must also be connected.

Since neither the computer nor the intelligent instrument is a modem, both are often configured as DTE. Then, the connections between the SLU and the instrument must be crossed to accommodate the directionality of each line. This has the effect of tricking each device into thinking that it is connected to a modem. Hence, a cable in which the data and handshake lines are crossed is called a *null modem* cable. The wiring diagram for a standard null modem cable is shown in Figure 7-4.

Note that some of the lines are jumpered on each end. RTS and CTS are jumpered so that whenever either device asserts RTS, it sees CTS asserted instantaneously and concurrently. This indicates that the "modem" (that is, the null modem cable) is ready to pass data to the remote device. By comparing the null modem connections with the functions of the lines described above, it should be possible to infer how the null modem cable imitates a true modem.

Unfortunately, this relatively simple plan is seldom followed by instrument manufacturers. Minimally, each device must be capable of signalling the other when it is able to receive data. Thus, each device must have dedicated lines which serve the following handshake functions:

Figure 7-4: Null Modem Cable Connections



- an output line which is asserted when the device's FIFO has room for more data
- an input line which it tests before sending data to the other device

These two lines must cross in the cable (assuming both devices are DTE). The problem is that no two devices seem to use the same pair of lines for this minimal handshake.

Based on the descriptions of the different handshaking lines given above, it may seem that the most likely pair for a minimal handshake is DTR and DSR. In fact, many intelligent instruments use this pair for handshaking. However, some instrument manufacturers apply the interpretation that Clear to Send should signal the readiness of the FIFO to accept data. They therefore instruct the user to build a cable in which, for example, CTS and RTS are crossed. Note that, in such a case, a standard null modem cable will not work.

Fortunately, most manufacturers are explicit in explaining how to set up hardware handshaking to their instruments. The user's manual will offer advice such as "...the signal on pin 4 is asserted when the Ajax Model 1040 is able to accept data..." or "...pin 3 should be wired to the signal line which the computer asserts when it is able to receive data..."

But what if the computer's SLU did not support hardware handshaking but the instrument's serial port did, for example? In this case, the signal lines which the instrument's serial port reads to detect the readiness of the computer to accept data should be tied to the instrument's Signal Ground line. This effectively disables handshaking by constantly "asserting" the handshake line. (Remember that when handshaking is disabled, the serial port becomes a realtime device!)

7.3.3 Cable Length

The line driver circuits in any serial device are only capable of supporting signal transmission over a finite distance. The allowable cable length depends on a number of factors, including the characteristics of the line driver circuits, the speed of transmission (that is, the baud rate), the gauge of wire used in the cable, and whether the cable is shielded or not. As a conservative rule of thumb, cables should be kept to under 3000 feet at 110 - 1200 baud and to under 250 feet at 2400 - 9600 baud.

If it is necessary to run a serial line farther than this, either a standard "acoustic" modem or a "short haul modem" can be used. Standard modems communicate over telephone lines, and thus can be used between any two locations with telephone service. Even over short distances, however, telephone lines may be noisy, causing errors in transmission. Short haul modems communicate via dedicated phone cables over distances of up to several miles (maximum distance differs among manufacturers).

Chapter 8

How To Use A Clock/Counter Module

A realtime clock/counter, as the name implies, is a multifunction module which can be used to measure time intervals or count signal pulses. Most clock/counter modules perform both timing and counting functions in a variety of modes. As examples, a single clock/counter module might be capable of performing each of the following functions (not simultaneously):

- Every 100th occurrence of a pulse, trigger an A/D converter.
- When a pulse occurs, wait 20 msec, then trigger an A/D converter.
- Measure the elapsed time between two pulses.
- When a pulse occurs, begin triggering A/D conversions at 2-msec intervals.
- Count the number of pulses on one channel between successive pulses on a second channel.

The mode of operation of a clock/counter module is determined by three mutually independent parameters: the source of the input signal, the nature of the output, and the mechanism for onset and termination of operation.

The *source* of the input signal determines whether the module is functioning as an interval timer or as a counter. Interval timing is accomplished by counting pulses from a constant frequency oscillator. When functioning as a counter, the module counts pulses from an external source. Thus, the only difference between the interval timing function and the pulse counting function is the source of the pulses being counted.

The *output* of a clock/counter module can take one of two forms:

- When a predetermined interval has elapsed or a predetermined count is exceeded, the clock/counter generates a signal which can be used to trigger another device directly or to interrupt the system.
- The elapsed time or number of pulses which occurred since the module was last reset can be read as a data value.

In the first case, the clock/counter is used to signal an event - the passage of a given amount of time or the occurrence of a given number of pulses. In the second case, the module is used to record the length of an indeterminate time interval or the number of pulses occurring in an interval.

Finally, the *onset and termination* of clock/counter operation can be controlled either by software (setting or clearing a bit in the CSR) or by an external trigger signal.

This chapter describes the operation and programming of Digital's KWV11-C clock/counter module. This particular module is widely used on Digital Q-bus systems and is similar in functionality to the KW11 (UNIBUS) and MNCKW (MINC) modules.

8.1 Clock/Counter Internal Operation

Like other realtime option modules, any clock/counter module has registers on the computer side which appear as storage locations in the I/O page and various signal pathways on the external side. In addition, these modules have various internal registers, oscillators, and other circuitry for conditioning external signals. The following sections describe these features and present an overview of the ways in which they work together in various operational modes.

8.1.1 Registers

The KWV11-C has two registers in the I/O page: a buffer/preset and a CSR register. The CSR is used to control the operational status of the module. The buffer/preset register is used to pass counts back and forth between the system and the counter.

The *counter* is an internal register which is not directly addressable; that is, it does not appear as a storage location in the I/O page. It can only be written to or read from indirectly through the buffer/preset register (see Section 8.2). The counter counts pulses originating either from the internal oscillator or an external source. Since it is a 16-bit register, it can hold values up to 65536 (unsigned). When that limit is exceeded, the counter overflows. On counter overflow, a flag bit in the CSR is set; an output pulse is generated; and several other things may happen, depending on the mode in which the module is operating. Thus, the counter register is central to the operation of the module.

8.1.2 Internal Circuits

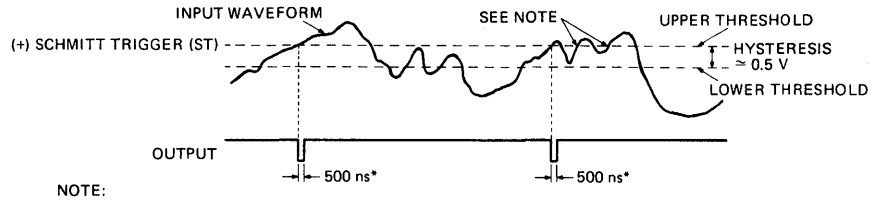
The KWV11-C has two on-board *Schmitt triggers* - ST1 and ST2. A Schmitt trigger is a circuit which detects threshold crossings in analog signals (see Figure 8-1). When the input voltage crosses a set threshold, a single, brief pulse is output. Both the threshold level and the direction of threshold crossing can be controlled, either by configuring jumpers and potentiometers on the module or through external connections.

The two Schmitt triggers are used in different ways. ST1 can be an external timebase input or an external input for signals to be counted. ST2 can be used to start the counter, to set a flag in the CSR, or to generate an interrupt to the CPU.

In addition, the KWV11-C has an internal oscillator circuit. This circuit produces pulses, or "ticks," at regular time intervals. The interval between ticks is selectable by setting certain bits in the CSR. Internal oscillator rates of 1 MHz, 100 kHz, 10 kHz, 1 kHz, or 100 Hz can be selected.

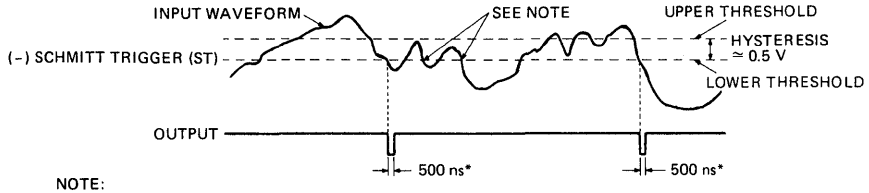
Either the internal oscillator or ST1 output pulses can be used as input to the counter register, depending on the setting of bits in the CSR.

Figure 8-1: Schmitt Trigger Operation



NOTE:
ST IS TRIGGERED AGAIN ONLY AFTER THE INPUT WAVEFORM DROPS BELOW THE LOWER THRESHOLD AND EXCEEDS THE UPPER THRESHOLD.

(a) POSITIVE SLOPE SELECTION (SLOPE SWITCHED ON)



NOTE:
ST IS TRIGGERED AGAIN ONLY AFTER THE INPUT WAVEFORM EXCEEDS THE UPPER THRESHOLD AND DROPS BELOW THE LOWER THRESHOLD.

(b) NEGATIVE SLOPE SELECTION (SLOPE SWITCHED OFF)

*400 ns MINIMUM

MR-5340
11-4549

8.1.3 External Connections

Externally, the KWV11-C has the following connections:

- Input and output connectors for each Schmitt trigger
- Slope control connectors for each Schmitt trigger
- External threshold level control connectors for each Schmitt trigger
- A clock overflow output (CLK OVF)
- A signal ground connection

Note that, in addition to input and output connections for the two Schmitt triggers, there are connections for setting the trigger slope and threshold levels. There are also various jumpers and potentiometers on the module itself for controlling the trigger slopes and thresholds. By setting jumpers on the module, you can select either this internal circuitry or external circuitry for Schmitt trigger control.

The clock overflow output can be led via the supplied external connector to a corresponding input connector on another module (analog or digital I/O or another clock module). In this way, the clock overflow signal can be used like an "external" trigger or clock input to other modules.

8.1.4 CSR Bit Assignments

The mode of operation of the KWV11-C is determined by the setting of various bits and bit-fields in the CSR. Figure 8-2 shows the bit assignments in the CSR. Each bit can be written or read under program control; however, certain bits have special programming considerations:

- The maintenance bits (8, 9, and 10) always read 0.
- The flags (7, 12, and 15) cannot be set by the program.
- The go bits (0, 13) can be cleared by more than one method.

Table 8-1 defines the function of each bit.

Figure 8-2: KWV11-C Control/Status Register Bit Assignments

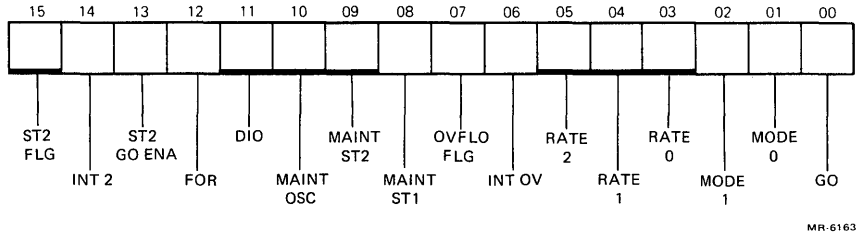


Table 8-1: KWV11-C Control/Status Register Bit Definitions

Bit	Name	Function	Set By/Cleared By
0	GO	Read/Write - Setting this bit starts the counter at a rate determined by the rate bits 3-5.	The GO bit is set and cleared under program control. In modes 1,2, and 3, this bit remains set until cleared by the program. In mode 0 this bit is cleared automatically when the counter overflows. Clearing bit 0 or a BUS INIT resets the counter and stops the counting.
1,2	MODE	Read/Write **2 1 Mode** 0 0 Mode 0 0 1 Mode 1 1 0 Mode 2 1 1 Mode 3	The mode is set and cleared under program control and by BUS INIT.
3-5	RATE	Read/Write - These bits select the clock rate or counting source for the counter. **5 4 3 Rate** 0 0 0 Stop 0 0 1 1 MHz 0 1 0 100 kHz 0 1 1 10 kHz 1 0 0 1 kHz 1 0 1 100 Hz 1 1 0 ST1 external input 1 1 1 Line (50/60 Hz)	The rate is set and cleared under program control and by BUS INIT.
6	INTOV (Interrupt on Overflow)	Read/Write - When this bit is set, the assertion of OVFLO FLAG generates an interrupt. Interrupt is also generated if bit 6 is set while OVFLO FLAG is set.	This bit is set and cleared under program control. If either bit 6 or 7 is cleared while an overflow interrupt request to the processor is pending, the request is cancelled.

Table 8-1 (Cont.): KWV11-C Control/Status Register Bit Definitions

Bit	Name	Function	Set By/Cleared By
7	OVFLO FLAG	Read/Write to 0 - If bit 6 is set, setting bit 7 generates an interrupt. Bit 7 must be cleared after the interrupt has been serviced to enable further overflow interrupts. If two enabled interrupts are requested at the same time by bits 7 and 15, bit 7 has the higher priority.	This flag is set each time the counter overflows. It is cleared under program control, or at the low-to-high transition of the GO bit, or by BUS INIT.
8	MAINT ST1	Write Only - Setting this bit simulates the firing of ST1. All functions started by ST1 can be exercised under program control by using this bit.	This bit is set under program control. Clearing is not needed. It is always read as a 0.
9	MAINT ST2	Write Only - Setting this bit simulates the firing of Schmitt Trigger 2. All functions started by ST2 can be exercised under program control by using this bit.	This bit is set under program control. Clearing is not needed. It is always read as a 0.
10	MAINT OSC	Write Only - For maintenance purposes, setting this bit simulates one cycle of the internal crystal oscillator used to increment the clock counter. (Bit 11 must be set.)	This bit is set under program control. Clearing is not needed. It is always read as a 0.
11	DIO (Disable Internal Oscillator)	Read/Write - For maintenance purposes, this bit prevents the internal crystal oscillator from incrementing the clock counter. This bit is used with bit 10.	This bit is set and cleared under program control.

Table 8-1 (Cont.): KWV11-C Control/Status Register Bit Definitions

Bit	Name	Function	Set By/Cleared By
12	FOR	Read/Write - Flag Overrun provides the programmer with an indication that the hardware is being asked to operate at a speed higher than is compatible with the software.	This flag is set when an overflow occurs and the OVFLO FLAG (bit 7) is still set from a previous occurrence, or when ST2 fires and the ST2 FLAG (bit 15) has been previously set. Bit 12 is cleared under program control, or at the low-to-high transition of the GO bit, or by BUS INIT.
13	ST2 GO ENABLE	Read/Write - When set, the assertion of ST2 FLAG sets the GO bit and clears the ST2 GO ENABLE bit.	The ST2 GO ENABLE bit is cleared under program control, or at the low-to-high transition of the GO bit, or by BUS INIT.
14	INT 2	Read/Write - When set, the assertion of ST2 FLAG (bit 15) causes an interrupt. If set while ST2 FLAG is set, an interrupt request is generated.	The bit is set and cleared under program control and by BUS INIT. When either bit 14 or 15 is cleared, any pending ST2 interrupt request is cancelled.
15	ST2 FLAG	Read/Write to 0 - Setting this flag starts an interrupt request if bit 14 is set. Bit 15 must be cleared after servicing an ST2 interrupt to enable further interrupts. If two enabled interrupts are requested at the same time by bits 7 and 15, bit 7 has the higher priority.	The ST2 FLAG is set by the firing of Schmitt Trigger 2 or the setting of the MAINT ST2 bit (in any mode) while the GO bit or the ST2 GO ENABLE bit is set. The ST2 FLAG is cleared under program control or at the low-to-high transition of the GO bit unless the ST2 GO ENABLE bit has previously been set. This bit is also cleared by BUS INIT.

Note that the same bit-field used to select the base clock rate is also used to select the source of pulses which drive the counter register.

8.2 Modes Of Operation

The setting of the mode bits (1 and 2) determines the basic operational mode of the module. The basic mode is like a theme on which many variations can be superimposed, depending on the setting of the other bits in the CSR. The following paragraphs give a brief description of each mode.

8.2.1 Mode 0 (Single Interval)

This mode of operation is used to generate a fixed count or a fixed interval for applications such as creating delay intervals.

1. The program loads the CSR with mode 0 (bits 1 and 2 both cleared), the clock rate, and interrupt enable (INTOV), if needed.
2. The program loads the buffer/preset register with the 2's complement (negative value) of the number of external events to be counted (when RATE = ST1), or the number of clock pulses needed to generate the time delay at the selected clock rate.
3. The program sets the GO bit (bit 0), or it sets the ST2 GO ENA bit (bit 13) and waits for an external event to set the GO bit.
4. When the GO bit is set, the counter is loaded with the contents of the buffer/preset register and starts counting.
5. The counter increments until it overflows, at which time it clears the GO bit and stops counting.
6. The overflow causes the OVFLO FLAG (bit 7) to be set in the CSR. If the INTOV bit was set, an interrupt occurs. If not, the KWV11-C waits for another program command.
7. The program either responds to the interrupt, or it checks the OVFLO FLAG (bit 7) to determine that the count or interval has been reached.
8. The program clears the OVFLO flag and, if no counting or mode changes are needed, sets the GO bit (or the ST2 GO ENA bit) to start again at step 4 above.

8.2.2 Mode 2 (Repeated Interval)

In this mode, the user can generate repeated intervals or counts. This mode is identical to Mode 0, with the following exceptions:

- When the OVFLO FLAG bit is set (step 6 above), the counter is automatically reloaded from the buffer/preset register, and counting continues without any additional program commands.
- If the counter overflows a second time before the program clears the OVFLO FLAG bit, the FOR bit (bit 12) is set to indicate a possible error condition. The counter is reloaded from the buffer/preset, and counting continues, as above.

Mode 2 is commonly used to generate a constant-frequency timebase.

8.2.3 Mode 3 (External Event Timing)

In this mode, the user can record the time of occurrence of external events, or count external events. Two external events can be monitored with respect to each other.

1. The program loads the CSR with mode 2 (bit 1 cleared, bit 2 set), the clock rate/source, and interrupt enable (INTOV or INT2), if needed.
2. The program sets the GO bit, or it sets the ST2 GO ENA bit and waits for an external event to set the GO bit. (NOTE: If both ST2 GO ENA and INT2 are set by a single program instruction, a race condition can occur, resulting in unpredictable operation).
3. When the GO bit is set, the counter is *cleared*, and starts counting.
4. When an ST2 pulse occurs, the contents of the counter are placed in the buffer/preset register and the ST2 FLAG bit is set. If INT2 was set, an interrupt occurs. Otherwise, the program can poll the ST2 FLAG bit to detect the occurrence of an ST2 pulse. The GO bit is *not* cleared.

5. The program can then read the buffer/preset register to determine the latency of the ST2 pulse or the number of ST1 pulses which occurred prior to the ST2 pulse (depending on the setting of the rate/source bits). The ST2 FLAG bit must be cleared after reading the buffer /preset to enable repeated operation.

6. If ST2 does not occur, the counter continues to increment even after an overflow. The overflow sets the OVFLO flag and generates an interrupt if INTOV is set.

7. Each successive ST2 pulse causes the contents of the counter to be placed in the buffer/preset register. If the ST2 FLAG is cleared when the ST2 pulse occurs, the ST2 FLAG bit is set. If the INT 2 bit is set, the setting of the ST2 FLAG bit causes an interrupt to be generated.

8. The counter continues to operate in this mode until the program clears the GO bit.

8.2.4 Mode 4 (External Event Timing From Zero Base)

This mode is identical to Mode 3 except that the counter is automatically cleared after every ST2 pulse (step 4 above). Thus, Mode 3 provides cumulative time/count information, whereas Mode 4 provides zero-based time/count information.

Chapter 9

Advanced Realtime Programming Techniques

Once you have successfully built the interface between your computer system and the real world, you face the job of programming the computer to perform the application. This chapter describes several programming techniques that you will find useful in programming realtime applications. Specifically, the following techniques are discussed:

- Buffer management techniques
- Direct device control on virtual systems
- Programming interrupts on virtual, multi-tasking systems

9.1 Buffer Management Techniques

Like many of the terms used in computer science, the word "buffer" can mean different things in different contexts. Originally, the term was applied to certain kinds of operations for controlling the flow of data in a system. For example, realtime data might be delivered to the system at a constant rate, while the CPU could process the incoming data only in spurts. During periods when the CPU was unavailable for processing, incoming data would have to be stored temporarily or it would be lost.

Consider the analogy of a pipeline carrying oil from a field of wells into a refinery. The wells pump oil out of the ground at a fairly constant rate, but the refinery processes the oil in spurts. To manage the flow of oil, a tank is used to hold the output of the wells until the refinery is ready to process it. At times, oil is removed from the tank for processing at a much faster rate than the rate at which the wells are pumping. At other times, the removal of oil from the tank is suspended completely. As long as the average rate at which oil is removed from the tank for processing is equal to the rate at which the wells are pumping, the system functions properly. The holding tank is analogous to a temporary storage area, or buffer, for data in a realtime computer system.

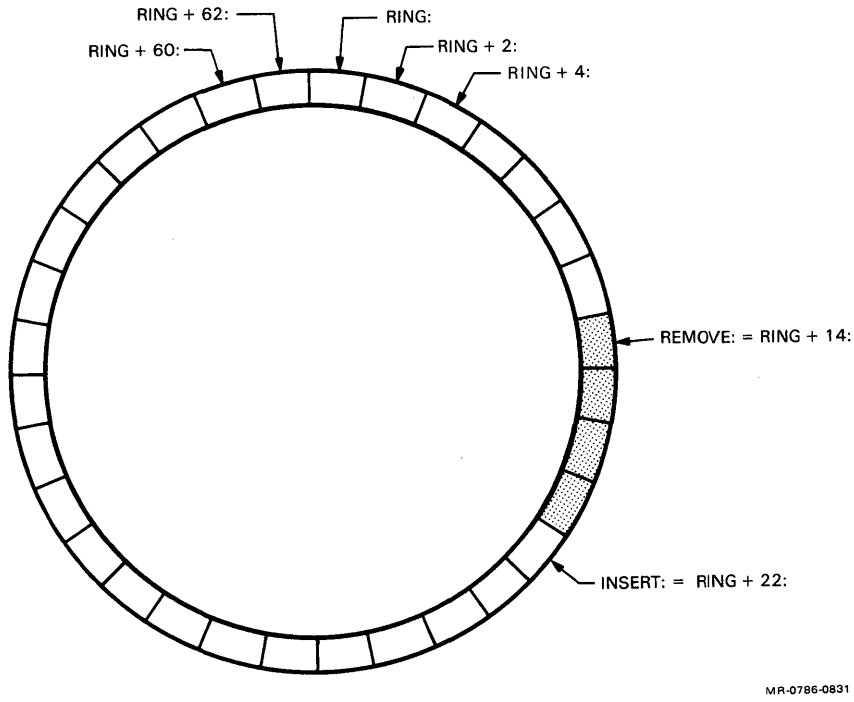
More generally, the term *buffer* can be applied to any temporary storage area for data, even when the temporal regulation of data flow is not a primary goal. For example, in a structured programming environment, you may want to handle data in functional blocks which are passed back and forth among program modules or processes.

This section presents information on basic concepts and techniques for storing data in buffers and for managing sets of buffers in realtime applications.

9.1.1 Ring Buffers

A *ring buffer* is a group of storage locations (that is, an array) into which data values are placed sequentially. After the last storage location is used, the next data value is placed in the first position in the array, and then the second, and so forth. Thus, the storage locations are continually reused. A pointer or index is used to keep track of the next element in the array into which data will be placed. Each time a new data value is stored in the array, the index is incremented. When the index becomes larger (by one) than the size of the buffer, it is reset to point to the first location in the array. This is shown diagramatically in Figure 9-1.

Figure 9-1: Diagramatic Illustration of a Ring Buffer



The program must remove data from the ring buffer before it is overwritten by newer data. In some cases, a second pointer is maintained which points to the next array element from which data is to be removed. Each time a data value is removed, this pointer is incremented and, like the input pointer, reset to point to the first element of the array when it exceeds the buffer size.

It is often convenient to use ring buffers whose length is an integral power of 2, since the pointers can be reset by merely zeroing the high-order bits after each input or output operation. This is shown in the code fragment shown in Example 9-1.

Example 9-1:

```
SUBROUTINE INSERT (VALUE)
INTEGER VALUE, POINTER, RING(1024)
COMMON /RINGBUFF/ RING
DATA POINTER /0/
POINTER = POINTER + 1
RING(POINTER) = VALUE
POINTER = POINTER .AND. 1023
RETURN
END
```

The constant 1023 (decimal) is equal to the binary value 0000001111111111. By logically ANDing the pointer with this constant each time it is incremented, the pointer is made to reset to 0 when it reaches 1024. Of course, this can also be accomplished by executing the FORTRAN statement:

```
IF (POINTER .EQ. 1024) POINTER = 0
```

However, the former method is more efficient, particularly when coding in assembly language.

9.1.2 Double- And Multi-Buffering Techniques

In many realtime applications, data can be processed in blocks, rather than on a point-by-point basis. Furthermore, each processing step often requires considerably less CPU time than the elapsed time needed to capture a block of data. In such cases, the use of two or more buffers to manage data can improve program efficiency dramatically.

Consider the example of an application in which 4 channels of analog data are continuously captured on a 100 Hz timebase (aggregate rate of 400 Hz) and written to a disk file. Assuming that no special processing of the data needs to be done in realtime, data can be handled in blocks of 400 data points. Each block represents 1 second of data. This example application can be decomposed into two discrete processing steps - a data-capture step and a disk-write step.

The application program must perform these two steps repeatedly and in sequence in such a way that no data are lost.

The code fragment in Example 9-2 illustrates a "brute force" approach to programming this example application.

Example 9-2:

```
INTEGER*2 IBUFF(400)
  .
  .
SECONDS = 1
DO WHILE (SECONDS .LE. 100)
  CALL GETDATA_S (IBUFF, 400)
  WRITE (1'REC1) IBUFF
  SECONDS = SECONDS + 1
END WHILE
  .
  .
```

The routine GETDATA_S is assumed to execute synchronously. That is, once it is called, control does not return to the main program until 400 data values have been captured and stored in the buffer (IBUFF).

The code sequence shown in Example 9-2 is logically valid, but it would fail to produce the desired result. This is because the disk-write and loop-control operations must be executed in the realtime interval between two successive sample times. At a clock rate of 100 Hz, this interval is less than 10 msec. If the system could not write the buffer to disk and loop back to call the sampling routine (GETDATA_S) in less than 10 msec, data would be lost. Stated differently, this technique would work for very low data acquisition rates, but not for even modestly high rates.

Double-buffering of data can improve the efficiency (and, thus, the performance) of realtime data acquisition applications. The following paragraphs describe how double-buffering could be used to perform the same application shown in Example 9-2.

If an asynchronous (interrupt-driven or DMA) data acquisition routine were used, the amount of CPU time required to capture 400 data values would undoubtedly be much less than 1 second. In that case, the left-over CPU time could be used to write a second block of data out to disk. The disk-write operation also is performed asynchronously by the disk driver. Since both of the required operations, data-capture and disk-write, can be executed asynchronously using less than 1 second of CPU time, the two can be superimposed as shown in Example 9-3.

Example 9-3:

```
      INTEGER*2 IBUFF1(400), IBUFF2(400), FLAG1, FLAG2
        .
        .
      FLAG1 = 0
      FLAG2 = 0
      CALL GETDATA_A (IBUFF1, 400, FLAG1)
      SECONDS = 1
      DO WHILE (SECONDS .LE. 100)
111     IF (FLAG1 .EQ. 0) GO TO 111
        CALL GETDATA_A (IBUFF2, 400, FLAG2)
        WRITE (1'IREC1) IBUFF1
        FLAG1 = 0
        SECONDS = SECONDS + 1
222     IF (FLAG2 .EQ. 0) GO TO 222
        CALL GETDATA_A (IBUFF1, 400, FLAG1)
        WRITE (1'IREC1) IBUFF2
        FLAG2 = 0
        SECONDS = SECONDS + 1
      END WHILE
        .
        .
```

The routine GETDATA_A is assumed to execute asynchronously. That is, when it is called, asynchronous data acquisition is enabled, and control returns to the main program. When data acquisition is complete, GETDATA_A sets the value of the third argument (FLAG1 or FLAG2) to 1.

The code sequence shown in Example 9-3 is very efficient in that it does not require a disk-write operation between the completion of one data-capture operation and the beginning of another. As soon as routine GETDATA_A signals that one buffer has been filled with data, the program starts data acquisition into a second buffer by calling GETDATA_A again. While GETDATA_A is filling one buffer with data, the buffer which was filled by the preceding call to GETDATA_A can be written out to disk. The system has a full second in which to write out a data buffer before the buffer is needed again for data acquisition.

Of course, the system must still satisfy the requirement that each successive call to GETDATA_A can reenable data acquisition within 10 msec of the completion of the previous data-capture operation. However, this is much more reasonable than having to perform the disk-write operation in that amount of time, as well.

The term *buffer chaining* is used to describe the operation of starting data acquisition into a new buffer when the previous buffer is filled. The present example application could be performed even more efficiently if buffer chaining were performed within the interrupt service routine which underlay the data-capture routine. To do this, the main program would have to pass the starting addresses of both buffers (IBUFF1 and IBUFF2) as well as the addresses of the two completion flags (IFLAG1 and IFLAG2) to the data-capture routine.

Another way to handle buffer chaining would be for the interrupt service routine to place data into an 800-word ring buffer. As the ring buffer was repeatedly filled, the interrupt service routine could set a flag at each 400-word boundary which signalled the main program to write out a block of data to disk. A technique similar to this is used in the programming example shown in Section 9.3 below.

Multibuffering of data can be used to achieve additional performance improvements under some circumstances. In the example application described above, suppose that the disk was heavily loaded due to some other activity on the system. Most of the time, the system might still be able to write a 400-word buffer of data out to the disk in under 1 second. Occasionally, though, it might take longer than 1 second to write out a buffer because of the overall loading of the disk. If only two buffers were used, the application would fail when that happened.

As long as the AVERAGE time required to write out a buffer of data is less than 1 second, the application can be successfully performed by using multiple buffers. A technique for performing the example application using multiple buffers is shown in Example 9-4.

Example 9-4:

```
        INTEGER*2 IBUFF(4,400)
        INTEGER*2 FLAG(4), BUFFN
        DATA FLAG /4*0/
          .
          .
        CALL GETDATA_M (IBUFF, 4, 400, FLAG)
        BUFFN = 1
        INDEX = 1
        SECONDS = 1
        DO WHILE (SECONDS .LE. 100)
111       IF (FLAG(BUFFN) .EQ. 0) GO TO 111
          WRITE (1'IREC1) (IBUFF(J), J = INDEX, INDEX+399)
          FLAG(BUFFN) = 0
          SECONDS = SECONDS + 1
          BUFFN = BUFFN + 1
          INDEX = INDEX + 400
          IF (BUFFN .EQ. 5) BUFFN = 1
          IF (INDEX .EQ. 1201) INDEX = 1
        END WHILE
          .
          .
```

The routine GETDATA_M is assumed to execute asynchronously. The main program passes arguments to GETDATA_M indicating the starting address of the data buffer (IBUFF), the number and length of subbuffers (4, 400), and the address of the flag array (FLAG). As each subbuffer is filled, GETDATA_M chains the next subbuffer in sequence and sets the value of the appropriate completion flag to 1. The main program tests each completion flag in sequence and writes the appropriate subbuffer of data out to disk when it finds a flag set.

The code sequence shown in Example 9-4 allows the system to perform the application even if, occasionally, a subbuffer cannot be written out to disk in under 1 second. As long as the maximum time required to write out 4 subbuffers of data is less than 4 seconds, the application will succeed.

9.2 Direct Device Control On Virtual Systems

The most basic level at which realtime devices can be controlled is by direct manipulation of the contents of device registers. Direct control of device registers is relatively simple in PDP-11 systems and, in fact, is a common programming technique for certain kinds of realtime applications. For example, the following MACRO-11 code fragment illustrates the acquisition of a single data value from an A/D device.

```
        ADCSR   = 770400        ;Address of device CSR
        ADBUF   = ADCSR+2       ;Address of device buffer
        MTPS    #340            ;Set processor priority level to 7
        MOV     #1,ADCSR        ;Set the GO bit to start a conversion
LOOP:   BIT     #200,ADCSR      ;Test the A/D DONE bit
        BEQ     LOOP            ;If clear, test again
        MOV     ADBUF,R0        ;If set, move the data value
                                ;  into register 0
        MTPS    #0              ;Drop priority level back to 0
```

In this example, the processor priority level is first raised to 7. This insures that the processor will execute the subsequent instructions without interruption. An A/D conversion is then initiated by setting bit 0 of the device control/status register (CSR). When the conversion is complete, bit 7 of the CSR is set by the device. The program repeatedly tests this bit until it finds the bit to be set, then moves the data value from the device's buffer register into a general-purpose processor register (or into a storage location in main memory). This technique is called "polled I/O".

Since many polled I/O operations require that the processor be totally dedicated to the polling loop, this technique has not been widely used on virtual, multiuser systems, such as VAX/VMS. However, the increasing use of MicroVAX systems for dedicated realtime processing has brought about a demand for direct device control routines. This section describes techniques for implementing direct device control routines on virtual, multi-tasking, multiuser systems using MicroVAX/MicroVMS as an example.

Note

As with RT-11 based PDP-11 systems, improper manipulation of device registers in the MicroVAX I/O page can lead to undesirable consequences. Generally speaking, direct device control techniques should be used to manipulate only process-dedicated realtime option module registers. Inadvertently changing other device register contents may cause a system crash or the corruption of a mass storage volume. Be sure to have a secure backup copy of the system whenever debugging direct device control routines.

To develop polled I/O (or other direct device control routines) under MicroVMS, two special programming techniques are called for:

- Addressing device registers in the I/O page
- Raising and lowering the processor priority level

This section illustrates the techniques used to accomplish these operations in the context of programs for performing single-channel clocked analog input using an AXV11-C analog module and a KWV11-C clock module.

9.2.1 Accessing Device Registers In The VAX I/O Page

On PDP-11 systems running the RT-11 Single-Job monitor, accessing addresses in the I/O page involves nothing more than supplying a 16-bit address in the range 160000 - 177776 (octal). Those addresses correspond to the portion of the physical address space within which device registers are defined - the I/O page. As with PDP-11 systems, the VAX architecture reserves a region of the physical address space for device registers. For MicroVAXes (I and II) the I/O page starts at address 20000000 (hex), and is 2000 (hex) bytes in length.

Under MicroVMS, the 32-bit virtual address space of the system is mapped into physical memory, so that a user-supplied address does not ordinarily correspond to any particular physical address. The problem, then, is to establish a means of addressing a particular range of physical addresses - that is, the MicroVAX I/O page.

This is accomplished by mapping the physical addresses which constitute the I/O page into a portion of the process's virtual address space. The VMS $CRMPSC system service is used. The following parameters are passed to the $CRMPSC service:

- The starting and ending virtual addresses into which the section is to be mapped. This virtual address block should be 8K bytes long and page-aligned.

- The starting page frame number of the section to be mapped (a page frame is a 512-byte block of physical memory). This is computed as <I/O page starting physical address>/512.
- The number of pages in the section (16 in this case).

In addition, the $CRMPSC system service accepts a flag mask specifying the type of section to be created, as well as its characteristics. In the present context, the essential flags are SEC$M_PFNMAP and SEC$M_WRT, which define the section as a page frame section with the read /write attribute.

Finally, the call to the $CRMPSC system service must specify storage locations into which the starting and ending virtual pages actually mapped, and the channel number assigned by the system to the section. Ordinarily, these data are not used by the programmer and need not be discussed further here. For a fuller discussion of the $CRMPSC system service, see the VAX/VMS System Services Reference Manual.

Once the mapped section has been created, instructions which address locations within that virtual address block effectively address the corresponding locations in the I/O page. To compute the virtual address of a particular device register in the I/O page, the absolute offset in the I/O page of the register is added to the base address of the virtual address block. This yields a virtual address which corresponds to the desired physical address.

9.2.2 Raising And Lowering Processor Priority Level

VAX architecture defines 32 interrupt priority levels (IPL 0 - IPL 31). At any point in time, the processor operates at some particular IPL, usually 0 during execution of user code. Device interrupts occur at IPL 20 - 23, corresponding to bus request levels 4 - 7. On MicroVAXes, driver code executes at IPL 23, regardless of the level at which the device interrupted. For example, a device interrupt on BR 4 is not granted until the processor priority falls below 20. However, when the request is granted, the processor priority is set to IPL 23, thus blocking ALL device interrupts until the driver lowers the IPL. The system timer interrupts at IPL 22, and is granted at the same level.

When the processor is executing code at a low IPL, device, timer, or software interrupts at a higher IPL can pre-empt the system. When this happens, the processor saves the context of the currently executing image (program counter, processor status longword) and transfers control to the interrupt service routine (ISR). When the ISR completes, the context of the pre-empted image is restored and continues executing where it left off (assuming no new interrupts are pending). Thus, an image executing at a low IPL can be suspended for various periods of time due to the occurrence of interrupts.

When performing polled I/O, you may want the processor to respond as quickly as possible to the availability of data from the device being polled. To prevent interrupts from distracting the processor from the polling operation, the polling code must be executed at an IPL above that of any device which might generate a pre-emptive interrupt. To accomplish this, the code raises the IPL to 30 using the DSBINT system macro. The value of 30 is chosen to enable a power-fail interrupt (IPL 31) to be processed, should one occur. This is advisable since allowing the power-fail ISR to execute could prevent corruption of the system device. Besides, blocking the power-fail ISR will probably not salvage the application.

VAX processor design imposes the restriction that IPL cannot be raised except while the processor is operating in kernel mode. The $CHMKNL system service must be invoked to change mode to kernel before executing the DSBINT macro. The polling code then executes in kernel mode. Whenever the processor is executing above IPL 2, page faults are fatal (the system will crash). Thus, before entering kernel mode, the $LCKPAG system service must be called to lock any data areas that are addressed in the polling routine into physical memory. When the polling routine completes, the program should restore the original IPL (usually 0), return to user mode, and, optionally, unlock the pages addressed in the polling routine.

In summary, the sequence of program steps in performing polled I/O at elevated IPL is as follows:

1. The I/O page is mapped into process virtual address space using the $CRMPSC system service.
2. Any required device initialization is performed at IPL 0, user mode.

3. Any pages addressed in the polling routine are locked into physical memory using the $LCKPAG system service.
4. The processor mode is changed to kernel using the $CHMKNL system service.
5. In kernel mode, the IPL is raised to 30.
6. The polling code executes and data is moved into a process buffer which has been locked into memory (step 3, above).
7. When data transfer is complete, the IPL is returned to its prior value (usually 0).
8. The processor mode is changed back to user mode by exiting the kernel mode routine.
9. The device is reset, if necessary.
10. Optionally, the pages addressed in kernel mode are unlocked using the $ULKPAG system service.
11. Post-processing of the acquired data is performed in user mode at IPL 0.

The following FORTRAN and MACRO-32 modules illustrate this sequence for single-channel clocked analog input using AXV11-C and KWV11-C modules (see Example 9-5).

Example 9-5:

To execute:

```
$ FORTRAN POLLED_AD
$ MACRO FASTAD32
$ LINK POLLED_AD,FASTAD32
$ SET PROCESS/PRIV=(CMKRNL,PSWAPM,PFNMAP) !Required privileges
$ RUN POLLED_AD
```

```
================================================================================

        PROGRAM POLLED_AD

        INCLUDE '($SYSSRVNAM)'

        INTEGER*2 IOFF, IVAL, IBUF(60000)
        INTEGER*4 ISTATUS, MAPIOP, NPNTS, LOCKS(2)

c  Use the $LCKPAG system service to lock the input buffer into physical
c  memory.

        LOCKS(1)=%LOC(IBUF(1))
        LOCKS(2)=%LOC(IBUF(60000))
        ISTATUS=SYS$LCKPAG(LOCKS,,)
        IF(.NOT.ISTATUS) CALL EXIT(ISTATUS)

c  Call routine MAPIOP to map the I/O page in virtual address space.

        ISTATUS=MAPIOP()
        IF(.NOT.ISTATUS) CALL EXIT(ISTATUS)

c  Input/initialize data acquisition parameters.

        TYPE 9060
9060    FORMAT('$Base clock rate, clock preset, number of samples? ')
        ACCEPT *, IRATE, KOUNT, NPNTS
        ICHAN = 0
        MODE = 0

c  Call the sampling routine. Control will return to the main program
c  only after I/O is complete.

        CALL FASTAD32(ICHAN,KOUNT,IRATE,IBUF,NPNTS,MODE,ISTATUS)

c  Output data, return status (residual AXV CSR)

        TYPE 9130, (J, IBUF(J),J=1,NPNTS)
9130    FORMAT(1X,2I10)
        TYPE 9140,ISTATUS
9140    FORMAT(/' ISTATUS =',O10)

        END

================================================================================
```

(Continued on next page)

Example 9-5 (Cont.):

```
        .TITLE FASTAD32
        .LIBRARY /SYS$LIBRARY:LIB.MLB/
        .PSECT   MAPIOP_MAIN    RD,WRT,PAGE
; NOTE: all subsequent MACRO-32 code in this example is contained in this
;       .PSECT.
VIOP:           .BLKB    8192           ; The virtual pages to which the
VIOP_END:                               ; I/O page will be mapped
PIOPAGE:        .LONG    VIOP           ; Starting and ending virtual page
                .LONG    VIOP_END       ;   addresses
RETPAGE:        .BLKL    2              ; Starting and ending page addrs
                                        ;   used (should be the same)
SECNAME:        .ASCID   /IOPAG_GLSEC/  ; Name of the section to be created
IOPAGECHAN:     .LONG                   ; Channel # to be associated with
                                        ;   the created section
PFNUM:          .LONG    ^X100000       ; Page frame number of the I/O page
                                        ;   (20000000 hex) / (200 hex)
;++
; Routine to map the I/O page into process virtual address space
;--
        .ENTRY  MAPIOP,^M<>
        $CRMPSC_S-
                INADR=PIOPAGE,-
                RETADR=RETPAGE,-
                FLAGS=#SEC$M_PFNMAP+SEC$M_WRT,-
                GSDNAM=SECNAME,-
                CHAN=IOPAGECHAN,-
                PAGCNT=#16.,-
                VBN=PFNUM
        RET
;++
;       FASTAD32 - Performs single-channel, polled I/O using AXV11-C and
;                  KWV11-C modules.
;
;       The FORTRAN calling interface is:
;
;       CALL FASTAD32(ichan,kount,irate,ibuf,npnts,mode,istatus)
;
;       where:
;               ichan   =       AXV11-C A/D channel number
;               kount   =       KWV11-C preset value
;               irate   =       clock rate:
;                                       1 = 1 MHz
;                                       2 = 100 kHz
```

(Continued on next page)

Example 9-5 (Cont.):

```
;                                       3 = 10 kHz
;                                       4 = 1 kHz
;                                       5 = 100 Hz
;               ibuf    =       array to store data
;               npnts   =       number of elements in ibuf
;               mode    =       if 0 then start immediately;
;                               if<>0 then start on ST2
;               istatus =       return status; if<0 then error
;
; This routine should be in the same .PSECT as the MAPIOP routine.
;
; The CLK OVFL pin on the KWV is strapped to the RTC IN pin on the AXV.
;
;--
        ADCSR   = VIOP+^O10400  ;address of AXV11-C A/D CSR
        ADBUF   = ADCSR+2       ;address of AXV11-C A/D BUFFER
        KWCSR   = VIOP+^O10420  ;address of KWV11-C CSR
        KWPRE   = KWCSR+2       ;address of KWV11-C BUFFER/PRESET
        ; Argument pointer offsets
        ICHAN   =  4
        KOUNT   =  8
        IRATE   = 12
        IBUF    = 16
        NPNTS   = 20
        MODE    = 24
        ISTATUS = 28

        .ENTRY  FASTAD32,^M<R6,R7,R8,R9,R10>
; Note that all instructions which address the I/O page are WORD MODE.
        TSTW    @#ADBUF         ;clear A/D DONE flag
        MOVZWL  @ICHAN(AP),R6   ;channel # to sample
        ASHL    #8,R6,R6        ;move channel # to byte 2 of R6
        BISB    #^O40,R6        ;enable clock driven
        MOVW    R6,@#ADCSR      ;load A/D CSR
        MOVZWL  @IRATE(AP),R6   ;clock rate in R6
        BICL    #^O177770,R6    ;clear excess bits
        ASHL    #3,R6,R6        ;shift rate to bits 3 - 5
        MOVW    R6,@#KWCSR      ;load KW CSR
```

(Continued on next page)

Example 9-5 (Cont.):

```
        MNEGW   @KOUNT(AP), -
                @#KWPRE         ;load KW preset register
        MOVL    IBUF(AP),R6     ;load address of IBUF into R6
        MOVZWL  @NPNTS(AP),R7   ;load NPNTS into R7
        MOVL    #ADCSR,R8       ;R8 points to A/D CSR
        MOVL    #ADBUF,R9       ;use R9 as pointer to A/D buffer reg
        MOVZWL  @MODE(AP),R10   ;pass MODE arg to POLL in R10
        $LCKPAG_S -             ;lock polling code into memory
                INADR=LCKPAG, -
                RETADR=LCKRET
        $CMKRNL_S  POLL                 ;execute routine POLL in kernel mode
        CLRW    @#KWCSR                 ;zero KWCSR - turns clock off
        MOVZWL  (R8), -                 ;return status is residual AXV CSR
                @ISTATUS(AP)
        CLRW    (R8)                    ;clear A/D CSR
        RET                             ;return to Fortran
LCKPAG: .LONG   POLL
        .LONG   POLL_END
LCKRET: .BLKL   2
        .ENTRY  POLL,^M<R6,R7,R8,R9,R10>
        DSBINT  #30                     ;disable all interrupts (except powerfail)
        TSTW    R10                     ;test mode
        BEQL    1$                      ;if 0 then trigger immediately
        BISW    #20002,@#KWCSR          ;set clock to wait for ST2
        BRB     2$                      ;and skip over next instruction
1$:     BISW    #3,@#KWCSR              ;trigger immediately
2$:     BBC     #7,(R8),2$              ;is conversion done?
        MOVW    (R9),(R6)+              ;store A/D value
        SOBGTR  R7,2$                   ;decrement NPNTS; if not zero, loop again
        ENBINT                          ;restore IPL to prior value
POLL_END:
        RET
        .END
```

(Continued on next page)

Example 9-5 (Cont.):

```
;++
;  These routines are not used in the present example. They are provided
;  only for general reference.
;
;  FORTRAN calling interface:
;
;     CALL IPEEK32 ( OFFSET, VALUE )
;
;     CALL IPOKE32 ( OFFSET, VALUE )
;
;        where:
;
;           OFFSET      = integer*2 offset in the I/O page
;
;           VALUE       = integer*2 value from/for the register addressed
;
; These routines should be in the same .PSECT as the MAPIOP routine.
;
;--
        .ENTRY  IPEEK32,^M<R6>
        MOVZWL  @4(AP),R6               ;put OFFSET in R6
        MOVW    L^VIOP(R6),@8(AP)       ;put value from iopage in VALUE
        RET

        .ENTRY  IPOKE32,^M<R6>
        MOVZWL   @4(AP),R6              ;put OFFSET in R6
        MOVW     @8(AP),L^VIOP(R6)      ;put VALUE into iopage
        RET
```

9.3 Interrupt Programming On Virtual, Multi-tasking Systems

Interrupt service routines on unmapped, single-tasking systems, such as PDP-11s running RT-11 are fairly straightforward. An interrupt occurs and control is transferred directly to the interrupt service routine (ISR). This involves saving the current program counter (PC) and processor status word (PSW) and replacing them with the PC and PSW stored at the vector address for the interrupting device. When I/O is complete, a completion flag can be set to notify the mainline program that the data have been stored (or output). The interrupt service routine returns control

to the mainline program by executing an RTI instruction which restores the saved PC and PSW registers. The mainline program may test the completion flag to detect I/O completion.

The virtual, multi-tasking, multiuser nature of VAX/VMS makes interrupt handling more complicated. Since the process which requested the I/O might not be current at the time an interrupt occurred, the ISR code and data must reside in system virtual address space and must execute in system context. If this were not the case, a context switch to that of the requesting process might be needed before the interrupt could be serviced. This would impose too great a penalty in response time. Similarly, notification of I/O completion must be handled asynchronously. Finally, the sharing of I/O resources among multiple, possibly noncooperating processes imposes additional architectural demands.

VMS resolves these issues through the implementation of an elegant, though complex, I/O subsystem. All VMS device drivers must interface to the I/O subsystem to insure the orderly flow of information between the various users' processes and shareable I/O devices. However, most realtime application programmers would prefer to avoid writing a full VMS device driver for controlling realtime I/O modules. This is particularly true in light of the fact that realtime devices generally need not be shareable among noncooperating processes, whereas device shareability is one of the sources of complexity in the I/O subsystem. The connect-to-interrupt driver enables users to program interrupt service routines and asynchronous I/O completion routines without having to write a full VMS device driver.

9.3.1 The Connect-to-Interrupt User Interface

The connect-to-interrupt driver (CONINTERR or CIN) is a template VMS device driver into which blocks of user-supplied code and data can be linked. The user-supplied code and data are contained in a single .PSECT hereafter referred to as the "CIN buffer". The CIN buffer is compiled and linked as part of the application program. The linkages between the CIN driver and the user-supplied CIN buffer are formed at run time by the $QIO system service. As a part of the process of building these links, the CIN buffer is mapped into system virtual address space (S0), while preserving the mapping in process virtual address space (P0). The result is that the CIN buffer is doubly-mapped in S0 and P0 virtual address space. Thus, code and data in the CIN buffer are accessible in both system and process context.

The CIN buffer comprises five sections:

1. A data area containing all data structures to be addressed during the execution of user-supplied CIN code.
2. A device initialization routine which is executed during recovery from a power failure.
3. A start I/O routine which is executed at the time the $QIO is issued.
4. An interrupt service routine which is executed in response to a device interrupt.
5. A cancel I/O routine which is executed when the user process issues a cancel I/O request.

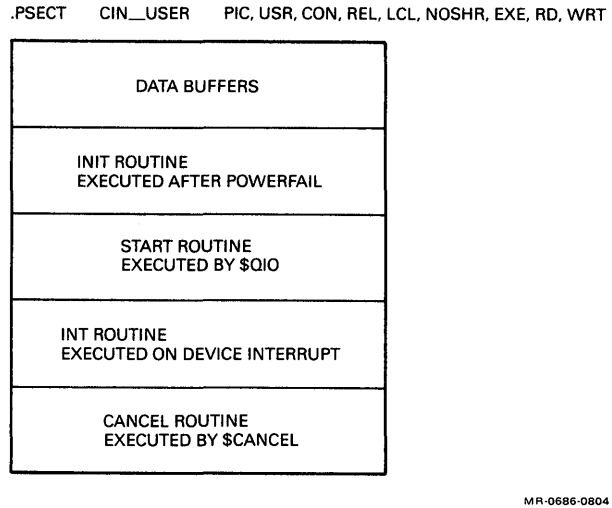
In addition, the user can specify an AST routine to be executed in process context on I/O completion (or partial completion). Note that any user-supplied code in the CIN buffer can only address data and code contained within the CIN buffer. Code which executes in process context, including the user-specified AST routine, can also address data and code in the CIN buffer and, of course, in any other portion of process virtual address space. The sections of the user-supplied CIN buffer are illustrated in Figure 9-2.

This application note is intended to provide an overview of CONINTERR concepts for intermediate to advanced programmers who want to get started using this facility. Many of the details of CONINTERR functionality and internals have been omitted, though what is presented is sufficient for many applications. For a more detailed description of CONINTERR, see the manual *Writing a Device Driver for VAX/VMS*, Appendix H.

9.3.2 Example Code Internals

The following example program performs continuous interrupt-driven analog input to a process buffer using a KWV11-C clock module and an AXV11-C analog module for timebase generation. Data are sampled into a 4096-word ring buffer with two subbuffers. The ring buffer is contained in the CIN buffer, and is therefore doubly mapped in S0 and P0. When a subbuffer is filled, an AST is delivered to the calling process. The AST routine moves the data from the subbuffer to a process buffer (singly mapped in P0 virtual address space). The AST routine then checks to determine whether I/O is complete - that is, whether the process buffer has received all the data requested. If it has, the AST issues a $CANCEL

Figure 9-2: The CIN Buffer



system service call to terminate I/O, and sets a completion flag to notify the calling program of I/O completion.

```
                SYSTEM SETUP FOR CONNECT-TO-INTERRUPT
1. Log in to SYSTEM account.
2. Insert the following line in the file SYS$SYSTEM:MODPARAMS.DAT
   REALTIME_SPTS = 20
3. Enter:
   $ @SYS$UPDATE:AUTOGEN SAVPARAMS REBOOT
        .
        .
   (system reboots)
4. Log in to SYSTEM account or other privileged account and enter:
   $ MCR SYSGEN
   SYSGEN> CONNECT AXA0:/ADA=0/CSR=%0770400/VEC=%0400/DRIVER=CONINTERR
   SYSGEN> EXIT
   $ MACRO AXV
   $ FORTRAN CALL_AXV
   $ LINK CALL_AXV,AXV
   $ SET PROC/PRIV=(PSWAPM, CMKRNL)
   $ SET PROCESS/PRIORITY=17
   $ RUN CALL_AXV
=========================================================================
```

System page table entries for double-mapping of CONINTERR buffers are drawn from a pre-allocated pool. In steps 1 - 3, above, the size of this pool is set by modifying the SYSGEN parameter REALTIME_SPTS. The number of page table entries allocated must be sufficient to map the user buffers (data and code) of all CONINTERR-driven devices which are connected at any given time. In the present example, 20 page table entries are allocated, sufficient to map 5120 words of data and code. This step needs to be taken only when additional REALTIME_SPTS are required for mapping CONINTERR buffers.

In step 4, above, the device is connected to the system. In this case, the device is given the name "AXA0"; this is the name used in the $ASSIGN system service call. Switches are used to designate the adapter number (always 0 on MicroVAXes), the CSR and vector addresses of the module, and the driver name (CONINTERR). This step needs to be taken each time the system is booted. The commands can be inserted into the file SYS$MANAGER:SYCONFIG.COM, if desired.

Example 9-6:

```
            PROGRAM CALL_AXV        !Calling program for AXV_SAMPLE
            INCLUDE '($SYSSRVNAM)'
            BYTE ICMPF
            INTEGER*2 PBUFF(30000)
            INTEGER*4 ISTATUS, ICHAN, IPRESET, ICOUNT
            INTEGER*4 LOCKS(2)
            INTEGER*4 AXV_SAMPLE
            COMMON /PROCESS_DATA/ PBUFF, ICMPF
c  Lock the process buffer into the working set. This is not essential,
c  but it helps to avoid page faulting in the AST routine.
            LOCKS(1) = %LOC(PBUFF(1))
            LOCKS(2) = %LOC(PBUFF(30000))
            ISTATUS = SYS$LCKPAG (LOCKS,,)
            IF(.NOT.ISTATUS) CALL EXIT (ISTATUS)
c  Assign a channel number for the AXV
            ISTATUS = SYS$ASSIGN ('_AXAO:' ,ICHAN,,)
            IF(.NOT.ISTATUS) CALL EXIT (ISTATUS)
c  Input/initialize arguments to be passed to the calling routine
            TYPE *, 'KWV preset, sample count?'
            ACCEPT *, IPRESET, ICOUNT
c  Routine AXV_SAMPLE handles all of the details of setting up and
c  executing the I/O operations. When the process buffer is filled,
c  the completion flag is set.
            ICMPF = 0   ! Clear the completion flag
            ISTATUS = AXV_SAMPLE (ICHAN, IPRESET, ICOUNT)
            IF(.NOT.ISTATUS) CALL EXIT (ISTATUS)
c  By looping on the completion flag, the process is insured of remaining
c  computable throughout the I/O operation.
100         IF(ICMPF.EQ.0) GO TO 100
c  For the sake of simplicity in this example, only a few of the acquired
c  data values are output. In a real application, the following step could
c  be replaced with file/graphic output.
        TYPE 9000, (J,PBUFF(J),J=1,1000,100)
9000    FORMAT(2I10)
        END
```

(Continued on next page)

Example 9-6 (Cont.):

```
        .TITLE      AXV
        .LIBRARY /SYS$LIBRARY:LIB.MLB/
        $IDBDEF                 ; Definition for I/O drivers
        $UCBDEF                 ; Data structurs
        $IODEF                  ; I/O function codes
        $CINDEF                 ; Connect-to-interrupt
        $CRBDEF                 ; CRB stuff
        $VECDEF                 ; more

        .PSECT PROCESS_ROUTINES     PIC,USR,CON,REL,LCL,NOSHR,EXE,RD,WRT
;++
; This example routine issues the QIO to connect to the AXV interrupts.
; It takes care of the internals associated with the connect-to-interrupt
; QIO.
;
; FORTRAN calling sequence:
;
;   STATUS = AXV_SAMPLE ( CHAN, PRESET, COUNT )
;
;   where:
;
;      CHAN           = longword channel # for device _AXA0
;
;      PRESET         = longword preset value for KWV
;                       (positive value <= 32K)
;
;      COUNT          = longword # of samples
;
;      STATUS         = longword return status of $QIO call
;
; In this example, the AXV and KWV CSRs are "firm-wired" for the
; following characteristics:
;
;
;       AXV - gain=1, RTC ENABLE, DONE INT ENABLE, channel=0
;
;       KWV - GO, mode=1, rate=1 (1 MHz)
;
; Ordinarily, they would be configured according to arguments passed
; to this routine from the calling program.
;
; The CLK OVFL pin on the KWV is strapped to the RTC IN pin on the AXV.
;
;--
```

(Continued on next page)

Example 9-6 (Cont.):

```
        .ENTRY   AXV_SAMPLE,^M<>
; These values are stored in process address space for use in the AST
; routine
        MOVL    @4(AP),AXV_CHAN          ; Channel # for $CANCEL
        MOVL    @12(AP),POINT_COUNT      ; Point count
; These values are stored in system address space for use in the CIN user
; start routine.
        MOVW    #^O13,KWV_CSR_VALUE      ; GO, MODE=1, 1MHz
        MOVW    #^O140,AXV_CSR_VALUE     ; RTC ENABLE, DONE INT ENABLE
        MNEGW   @8(AP),KWV_PRESET_VALUE  ; Negated KWV preset value
       $QIO_S   CHAN=@4(AP),-            ;Channel
                FUNC=#IO$_CONINTWRITE,-  ;Allow writing to the data buffer
                IOSB=AXV_CIN_IOSB,-      ;I/O status Block
                P1=AXV_CIN_BUF_DESC,-    ;Buffer descriptor
                P2=#AXV_CIN_ENTRY,-      ;Entry list
                P3=#AXV_CIN_MASK,-       ;Status bits,etc
                P4=#USER_AST,-           ;AST service routine
                P6=#10                   ;preallocate some AST control
                                         ; blocks
       RET                               ;Return to calling routine
AXV_CIN_BUF_DESC:                        ;Buffer descriptor for CIN
        .LONG   USER_END - RINGBUF
        .LONG   RINGBUF
AXV_CIN_ENTRY:
        .LONG   USER_INIT - RINGBUF      ;Init code
        .LONG   USER_START - RINGBUF     ;Start code
        .LONG   USER_INT - RINGBUF       ;Interrupt service routine
        .LONG   USER_CNCL - RINGBUF      ;I/O cancel routine
AXV_CIN_IOSB:
        .LONG   0                        ; I/O Status Block
        .LONG   0
; Control mask - see VMS Release Notes, Appendix C for explication
AXV_CIN_MASK = CIN$M_REPEAT!CIN$M_START!CIN$M_ISR!CIN$M_CANCEL
```

(Continued on next page)

Example 9-6 (Cont.):

```
        .SBTTL  USER_AST, User AST routine
;++
; This routine is invoked when I/O completion is signaled by the USER_INT
; routine. It is queued to the user's process in the access mode of the
; $QIO which initiated the I/O. It executes in process context at
; IPL$_ASTDEL (IPL 2).
;
; I/O completion is signalled by loading SS$_NORMAL (1) into R0 on exiting
; the USER_INT routine. In the present example, this does not signal full
; completion of the I/O, since the CIN$M_REPEAT flag was set in the $QIO
; call.
;
;--

        .ENTRY  USER_AST,^M<R2,R3,R4,R5>
        MOVAL   RINGBUF,R2              ; Get CIN buffer address
        MOVZWL  RINGBUF_INDEX,R3        ; Get buffer index
        BBS     #11,R3,10$              ; Is bit 11 set?
        ADD     #4096,R2                ; No, xfer the top half
10$:    MOVAL   PBUFF,R4                ; Get process buffer base addr
        ADDL    PBUFF_INDEX,R4          ; Add index
        MOVC3   #4096,(R2),(R4)         ; Move data (trashes R0-R5)
        ADDL    #4096,PBUFF_INDEX       ; Update process buffer index
        SUBL2   #2048,POINT_COUNT       ; Update point count
        BGTR    20$                     ; Have we moved all the data?
        JMP     ALL_DONE                ; Yes - Finish up
20$:    MOVZWL  #SS$_NORMAL,R0          ; Set return status
        RET                             ; And return

ALL_DONE:
        $CANCEL_S -                     ; Cancel the I/O request
                CHAN=AXV_CHAN
        CLRL    PBUFF_INDEX             ; Reset index into PBUFF
        MOVB    #1,ICMPF                ; Set the completion flag
        MOVZWL  #SS$_NORMAL,R0          ; Set return status
        RET                             ; And return

PBUFF_INDEX:
        .LONG   0

POINT_COUNT:
        .LONG    0

AXV_CHAN:
        .LONG    0
```

(Continued on next page)

Example 9-6 (Cont.):

```
        .PSECT PROCESS_DATA      PIC,OVR,REL,GBL,SHR,NOEXE,RD,WRT,LONG
;++
; This is the FORTRAN common block /PROCESS_DATA/
;--
PBUFF:    .BLKW    30000
ICMPF:    .BYTE    0
          .PSECT CIN_USER            PIC,USR,CON,REL,LCL,NOSHR,EXE,RD,WRT
;++
; This is the CIN buffer. In the present example, it is contained in
; the same source code module as the process routines shown above.
;--
          .SBTTL   DATA STRUCTURES
DATA_BUF_SIZE  =      4096
RINGBUF:              .BLKW    DATA_BUF_SIZE
RINGBUF_END:
RINGBUF_INDEX:        .WORD
AXV_CSR_VALUE:        .WORD
KWV_CSR_VALUE:        .WORD
KWV_PRESET_VALUE:     .WORD
; In the CIN user routines, the system virtual address of the CSR of the
;   device is supplied. Other addresses in the I/O page used by the code
;   must be handled as offsets from the device CSR.
AXV_ADDRESS      = ^O170400                  ;Address of AXV
KWV_ADDRESS      = ^O170420                  ;Address of KWV
KWV_OFFSET       = KWV_ADDRESS-AXV_ADDRESS   ;Offset for KWV CSR
PRESET_OFFSET    = KWV_OFFSET+2              ;Offset for KWV BPR
DBR_OFFSET       =2                          ;Offset for AXV DBR
        .SBTT     USER_INIT        ; Dummy dev intialization routine
;++
; This routine is invoked after power recovery. It executes in system
; context at IPL$_POWER (IPL 31).
;
; This routine is not implemented in the present example. However, the
; entry point must be defined and included in the entry list of the
; $QIO (P2).
;
; See VAX/VMS Release Notes, Appendix C for inputs.
;
;--
```

(Continued on next page)

Example 9-6 (Cont.):

```
USER_INIT::
        RSB

        .SBTTL   USER_START, Start I/O routine

;++
; This routine is invoked by the $QIO system service. It executes in
; system context at IPL$_QUEUEAST (IPL 6).
;
;   On entry:
;
;       0(R2)  - arg count of 4
;       4(R2)  - Address of the process buffer (system mapped)
;       12(R2) - Address of the device's CSR
;
; See VAX/VMS Release Notes, Appendix C for other inputs.
;
; The routine must preserve all registers except R0-R4.
;
;--
CSR_ADD = 12                    ; Argument list offset of CSR addr
                                ; (only valid in USER_START and USER_INT)
USER_START::
        CLRW      RINGBUF_INDEX             ; Clear ring buffer index
        MOVL      CSR_ADD(R2),R0            ; Get address of the AXV CSR
        TSTW      DBR_OFFSET(R0)            ; Clear AD DONE bit, if set,
                                            ;   by reading AXV DBR
        MOVW      AXV_CSR_VALUE,(R0)        ; Set up the AXV
        MOVW      KWV_PRESET_VALUE, -       ; Set KWV preset
                  PRESET_OFFSET(R0)
        MOVW      KWV_CSR_VALUE, -          ; Set KWV CSR
                  KWV_OFFSET(R0)
        MOVZWL    #SS$_NORMAL,R0            ; Load a success code into R0.
        RSB                                 ; Return

       .SBTTL     USER_INTERRUPT, Interrupt service routine

;++
; This routine is invoked on device interrupt. It executes in system
; context at IPL$_DIPL (IPL 23 in MicroVMS).
;
```

(Continued on next page)

Example 9-6 (Cont.):

```
; On entry:
;
;       0(R2)  - arg count of 5
;       4(R2)  - Address of the process buffer
;       8(R2)  - Address of the AST parameter
;       12(R2) - Address of the device's CSR
;
; See VAX/VMS Release Notes, Appendix C for other inputs.
;
; The routine must preserve all registers except R0-R4
;
;--
USER_INT::
        MOVL      CSR_ADD(R2),R0            ; Get AXV CSR address
        MOVAL     RINGBUF,R1                ; Get CIN buffer address
        MOVZWL    RINGBUF_INDEX,R3          ; Get buffer index
        MOVW      DBR_OFFSET(R0),(R1)[R3]   ; Read data (assume no error)
        INCW      R3                        ; Increment buffer index
        BICW      #^O170000,R3              ; Clear all but bottom 12 bits of
                                            ;   (ring) buffer index
        MOVW      R3,RINGBUF_INDEX          ; Put updated index in storage
        CLRL      R0                        ; Clear return status
        BICW      #^O174000,R3              ; Now test bottom 11 bits of
                                            ;   buffer index
        BNEQ      10$                       ; Skip AST if not zero
; The user AST will be queued if the LSB of R0 is set on return
5$:     MOVZWL    #SS$_NORMAL,R0            ; Queue the AST
10$:    RSB
        .SBTTL    USER_CANCEL, Cancel I/O routine
;++
; This routine is invoked by the $CANCEL system service. It executes in
; system context at IPL$_QUEUEAST (IPL 6).
;
; On entry:
;
;   JSB interface:
;
;       R3      -  Addr of current IRP
;       R4      -  Addr of PCB of cancelling process
;       R5      -  Addr of the UCB
```

(Continued on next page)

Example 9-6 (Cont.):

```
;
;   CALL interface:
;
;       0(AP)       Arg count 4
;       8(AP)       Addr of IRP
;       12(AP)      Addr of PCB
;       16(AP)      Addr of UCB
;
; See VAX/VMS Release Notes, Appendix C for other inputs.
;
; The routine must preserve all registers except R0 and R3.
;
;--
USER_CNCL::
        MOVL    UCB$L_CRB(R5),R0          ; Get Address of the CRB
        MOVL    CRB$L_INTD+VEC$L_IDB(R0),R0 ; Address of the IDB
        MOVL    IDB$L_CSR(R0),R0          ; Get addr of AXV
        CLRW    KWV_OFFSET(R0)            ; Stop clock
        TSTW    DBR_OFFSET(R0)            ; Clear AD DONE bit
        CLRW    (R0)                      ; Clear AXV CSR
        MOVZWL  #SS$_NORMAL,R0            ; Load return status
        RSB                               ; And return
; Label that marks the end of the module
USER_END:                                 ; Last location in module
        .LONG
        .END
```

Chapter 10
Realtime System Performance

Performance in realtime applications is usually characterized by throughput - the number of operations performed per unit time - and response time - the interval required for the system to respond to an external event. Optimizing throughput or response time can be quite complex, particularly when very high performance relative to system capabilities is required. Selecting a system and peripherals with adequate realtime features can drastically simplify application development.

System realtime performance is determined predominately by the following factors:

- System bus structure and aggregate bandwidth
- CPU design and speed
- Operating system characteristics
- I/O controller speed and features

The relative importance of each of these factors depends heavily on the requirements of the specific application. Understanding how these factors interact to determine performance can help in selecting the best system and options for your realtime application.

10.1 System Bus Structure And Bandwidth

Most computers have a system bus servicing the CPU and main memory. In smaller computers, this is typically the only bus, and it supports all I/O operations. More powerful systems provide separate I/O buses that can be configured to channel I/O operations, rather than having external devices connected directly to the system bus.

The system bus determines maximum aggregate throughput for the computer, and has a rated or estimated total bandwidth. If your application's throughput requirements exceed the system's bandwidth, then you cannot perform the application. However, even if the rated system bandwidth is in excess of your application requirements, you must recognize that this bandwidth is NOT all available for I/O to realtime devices. Part of the system bandwidth is used to service system operations (for example, context switching, paging) and non-realtime devices. You must therefore evaluate these loads on the system as well as realtime loads to determine if the system has adequate bandwidth. Even if the system is to be totally dedicated to realtime operations, you should assume that only 50-75% of the rated bandwidth can be achieved in practice, unless you have specific knowledge or experience to the contrary.

Once you have selected a system with adequate bandwidth, use I/O buses to optimize data flows into or out of the system. On a VAX-11/785, for example, you can configure a high-speed input device on a separate Unibus Adaptor from mass storage devices that store the inputs in real time. If necessary, another Unibus Adaptor can be used for terminal I/O, printing, or networking operations. This approach allows you to access as much of the VAX-11/785's system bandwidth as you require for your application.

Table 10-1 summarizes the bus structure and bandwidth of Digital computers available today. Please note that the rated bus bandwidths are calculated taking into account overhead operations for bus protocol, and are approximate values only. The aggregate bandwidth reflects the maximum potential throughput on the bus from all operations concurrently. The single device value indicates the maximum sustainable rate from one device on the bus.

Table 10-1: Bus Bandwidth of Digital Systems

I/O Bus	Computers	Address Lines	Data Lines	Maximum Data Rate
UNIBUS	VAX 8600, 8650 PDP-11/24 PDP-11/44 PDP-11/84	18	16	1.5MB/s
Q-Bus	MicroVAX MicroPDP-11	22	16 (multiplexed address lines)	3.3MB/s
CTI	Professional 300 Series	22	16 (multiplexed address lines)	2.3MB/s
VAXBI	VAX 8200, 8300 8500, 8550 8700, 8800	30	32	13.3 MB/s

When do you need another I/O bus? Once the devices on the bus generate an I/O load equal to approximately 50% of the rated bandwidth, adding another bus adaptor will probably increase overall performance or simplify the work required to reach a given level of performance.

On smaller systems, the same loading rules of thumb apply, but no I/O buses are typically available. You connect devices directly to the system bus. If your planned realtime operations approach 50% of the rated bandwidth of the system, you should consider purchasing a higher performance system. This will probably be less expensive in the long run than trying to achieve the same level of performance in a smaller system through special programming or I/O options.

10.2 CPU Design And Speed

CPU design and speed are critical if your application involves program-controlled I/O or realtime processing. In program-controlled I/O, the CPU moves every word of data into or out of memory. You can poll the device (i.e., check the status of a bit or bits) to determine when each word is ready to be moved into or out of memory; or, if the processor design allows vectored interrupts, you can use them to let the processor know when each word is ready for input or output. The speed of the processor and its interrupt-handling logic determine (along with other application-dependent software factors) how fast data can be moved in this fashion.

If your application uses Direct Memory Access (DMA), then CPU speed is much less critical for I/O operations, since the CPU is involved only during arbitration for the bus by the DMA device. During data transfers, the DMA controller actually moves blocks of data while the CPU is free to perform other tasks.

If your application requires processing of incoming data in real time, however, then CPU performance is likely to be the factor constraining overall throughput. Simple processing operations (limit testing, summation, etc.) can be accomplished quickly. But, if you must perform a series of operations, then processing time is likely to be much longer than I/O time. A higher speed CPU, possibly coupled with a special purpose accelerator (for example, floating-point, array processor) may be necessary to meet your needs.

To determine processing throughput, you can either calculate the total time taken by a CPU to perform the specific operations, or you can run a benchmark and measure the elapsed time empirically. Direct calculation is practical only if your processing operation is very simple. For example, if you are polling a device to acquire data, your program might be limited to checking the status bit, moving the word, checking for a termination condition, incrementing a counter, and looping back to the status check. In such circumstances, you can actually add the instruction execution times of the few instructions required to estimate the cycle time per word of I/O. For more complex programs or programs written in a high-level language, however, benchmarks are usually necessary.

10.3 Operating System Characteristics

Operating system characteristics play a major role in determining realtime performance. By providing high-level services - such as multiple task scheduling, software priority levels, and I/O request management - the operating system can make inherently complex realtime tasks simple to program. However, these features and services involve operations that may not be necessary for your application - exception handling, error checking, allocation and verification of system resources, and the like. Services that you don't need are "overhead," and you may want to bypass them in the interests of optimized performance.

Four aspects of operating system design affect realtime programming most significantly:

- Task scheduling
- I/O programming services
- Intertask communication and synchronization
- User controls for optimizing performance

10.3.1 Task Scheduling

The number of users and tasks an operating system is required to handle concurrently plays a major role in determining the responsiveness and, in some cases, the throughput of the system. Single-user, single-task operating systems can be extremely small and simple; they require very little overhead. Digital's RT-11 is a premier example of such a small, simple, and efficient operating system.

If your application involves multiple events or processes occurring in parallel, however, you need a multi-tasking operating system to avoid complex systems programming. The operating system should allow you to set relative priorities among the tasks to determine when they will execute, rather than offering only the round-robin scheduling associated with multiuser timesharing systems. Without priority scheduling, your most time-critical tasks can fail to execute when they are needed. The system must also be able to respond to external events based on their priorities.

Multi-tasking operating systems inevitably impose more "overhead" on a single task, even if it is given the highest available priority. When it executes, the system scheduler uses resources that could be available to the application if no scheduling were required. Thus, in general, program-controlled I/O operations execute faster under a single-task operating system than a multi-tasking one. When multiple tasks are contending for the system, context-switching and related operations slow down both throughput and response times in a multi-tasking system. Compensating for this overhead, however, is the capability to handle many tasks and/or users concurrently.

While priority-driven multi-tasking systems can concurrently handle both realtime device interactions and interactions with multiple human users under appropriate circumstances, the potential conflicts should be recognized and anticipated. If, periodically, a direct conflict occurs between

time-critical device interaction and time-flexible user interactions, the human users must wait. This may cause disappointment or frustration, but the time-critical operation can be successfully completed. When this problem occurs, the only alternative is to use separate systems for the two types of interaction. This tends to be more expensive and ,in many situations, more cumbersome.

10.3.2 I/O Programming Services

The I/O programming services provided by an operating system determine how easily and efficiently I/O operations can be requested and executed. There is no single "right" approach, but rather several levels of programming and several modes of I/O operation that meet differing application needs.

I/O operations can be either synchronous or asynchronous with respect to the calling task. In synchronous I/O, the program requests an I/O operation and then waits for that operation to complete before it continues. This is appropriate if the program requires the results of the operation in order to proceed. Otherwise, synchronous I/O introduces substantial inefficiencies, since the I/O operation typically does not fully utilize system resources. In asynchronous I/O, the program requests an I/O operation and then continues to execute while the I/O operation is performed in parallel. This makes better use of the system resources by supporting more than one concurrent task. Asynchronous I/O also greatly facilitates vital operations such as realtime graphics or data processing by allowing the CPU to work on data as they are acquired rather than after the I/O operation is over. All Digital's realtime operating systems support both modes of I/O, even the RT-11 Single-Job Monitor - a single-task environment.

I/O programming level here refers to the degree of generality and simplicity involved in requesting an operation. The top level is generally a device-independent service providing access to system resources from high-level languages and database managers. Such device-independent services are often file- or record- oriented (rather than byte-oriented), and use standard formats to manage data. These services are easiest to use since they manage many aspects of data access and modification transparently. However, they incur excess overhead for users who do not need all the features they provide. All Digital operating systems have one or more device-independent file managers. The Record Management Service (RMS) package is also available for compatible I/O programming across the Professional Series, PDP-11s, and VAX/VMS systems.

At the next lower level is the I/O request manager that actually posts requests for I/O operations to device drivers and signals completion when the operation finishes. Higher level services - both file managers and other layered user interfaces such as subroutine libraries - typically make use of this request manager to perform their work as well. Several different services may be offered, depending on the modes of I/O supported (see Section 10.4, below). This level can include many precoded operations - for example, error checking, exception handling, and verification of requested resources - to simplify the user's job. Such operations prevent user mistakes from hanging or crashing the system, but also create latency periods after a request is issued from a user task until I/O actually starts. Thus, they impose overhead on the I/O operation.

The Queue I/O system service handles the I/O request manager function under VMS, RSX-11, and P/OS on Digital systems. RT-11 has its own I/O request manager. Using these services directly is a bit more difficult than working with RMS or a high-level subroutine calling interface, but it can be done from most high-level languages and does not involve detailed knowledge of the system or the I/O device.

The next level down is the actual device driver or handler itself. Many realtime devices are highly specialized and may not have device drivers available. Also, many users prefer to write their own device drivers to optimize performance by omitting nonessential operations which a general-purpose driver should have. (Sophisticated error-handling routines and multiple termination condition checking are good examples.) Specialized drivers can thus reduce overhead in the I/O operation itself, but they still involve the Queue I/O system service overhead described above.

All of Digital's realtime operating systems allow users to write and install their own device drivers. All of the documentation sets include explicit instructions for designing and building such programs. This usually involves programming in assembly language (although not in VAX/ELN or MicroPower/PASCAL), and thus requires user knowledge of the CPU instruction set as well as the I/O device's detailed design.

The lowest level of I/O programming service is direct access to the device itself. The user sets and clears bits in the device's CSRs and moves data into or out of memory as required by the device design. This level involves the minimum amount of operating system overhead, because the user is controlling all aspects of the I/O operation and need not incur delays from unwanted activities. Depending on how the approach is implemented, however, and what operating system is

employed, some overhead can still be present. You may have greater or lesser amounts of difficulty in accessing device registers and preventing unwanted operations.

10.3.3 Intertask Synchronization And Communication

Tasks that are executing concurrently must be able to communicate or synchronize with each other to provide maximum benefit. For example, a processing or graphics task must be able to recognize when data are ready in an input buffer; a disk-writing task must recognize when the processing task is complete and results are ready to be stored, etc. Four major methods are used for intertask communication:

- Asynchronous system traps
- Event flags
- Message systems
- Shared memory/disk areas

Asynchronous system traps (ASTs) respond to an external event that generates an interrupt by halting the executing task and executing a user-written service routine. They provide high-speed response to external events.

Event flags can be used to signal asynchronous as well as synchronous events. A task can set an event flag to signal completion to another task, or it can wait for a flag to be set before proceeding. Some operating systems also allow tasks to recognize events involving a combination of flags, such as logical ANDs or ORs. Event flags are handled with standard operating system services and have slower response times than ASTs.

Message systems can be managed by the operating system or by the user. In VMS, for example, mailboxes provide a message service managed by the system; while in RSX and RT-11, messages from one task to another are supported, but must be managed to a greater extent by the user.

Shared memory or files allow tasks to receive information from a common data base in memory or on disk. They can be read-only as in RSX, or read/write as in VMS. Access typically requires synchronization using event flags, lock management, or other tools.

10.3.4 User Controls

Optimizing realtime performance on any computer system is ultimately a matter of user control over system resources. No operating system meets all user needs completely; performance monitoring and system tuning are often essential.

All of Digital's realtime operating systems provide complete control over system resources. You can select high-level services that are useful in certain areas, and eliminate services in other areas that impose overhead. For example, VMS need not be a virtual operating system. You can lock tasks into memory and eliminate paging altogether, or lock some tasks in memory while allowing others to swap. Thus, you can have the benefits of virtual addressing while eliminating most of the potential overhead.

Often, the need for such control will influence your decision to program the application or the system at a low level. This is a fundamental trade-off; the only way to eliminate overhead may be to do without a service. Yet the ability to bypass services is not always provided, and many so-called realtime operating systems do not provide the services to begin with.

Digital's realtime operating systems give you a wide range of choices in the services available and also let you bypass those you don't need.

10.4 I/O Controller Speed And Features

Once you have selected a system with a CPU and buses which are adequate for your needs, the specific features of the I/O controller you use will have the greatest impact on realtime performance. The four key aspects of controller design that you should consider are:

- Choice of DMA or programmed I/O
- Types of I/O supported
- Special functions of the module
- Buffer memory

10.4.1 DMA Vs. Programmed I/O

I/O controllers with DMA capabilities can provide dramatic increases in throughput over programmed I/O (PIO) devices, as well as off-loading the CPU for other operations. Maximum data rates using PIO usually range from 1 to 20 kHz depending on CPU speed, operating system, and application characteristics. With DMA, the same CPU and operating system can support data rates from ten to several hundred kHz.

However, DMA operations can offer such performance enhancements only if the application involves transfers of relatively large blocks of data either into or out of the system. For applications that involve only one or two words per transfer, or require mixed inputs and outputs of a few words per operation, DMA offers little or no advantage over program-controlled I/O. Moreover, the hardware to support DMA can be expensive and may not be required for your application.

10.4.2 Types Of I/O Supported

The types of I/O operations supported by a peripheral can determine its maximum performance regardless of the system on which it is used. For example, very high resolution or wide dynamic range analog-to-digital conversion cannot be performed at the same speed as lower-resolution conversion without paying a significant dollar premium for such performance. If your application requires an inherently lower-speed form of I/O, then you can probably perform it with a less expensive system and I/O controller than a higher-speed application.

Some I/O controllers support multiple types of devices concurrently. For example, the AXV-11C performs both analog input and outputs. In such cases, you should be careful to investigate whether the controller supports all types of operations concurrently, and what impact concurrent operations will have on aggregate throughput. The throughput when performing multiple types of I/O is nearly always lower than when performing a single type of I/O because of software overhead.

The individual I/O ports of a controller or peripheral device usually have speed or throughput specifications that reflect hardware limitations. These specifications do not reflect software constraints, and may not be achievable easily, if at all. For example, the ADV11-C A/D converter is specified to have a maximum conversion rate of 25 kHz. However, since it is a program-controlled device, achievable data rates without special assembly language programming are in the 1 to 5 kHz range. Unless the

device specifications explicitly account for software factors, you cannot assume that the maximum hardware speeds are applicable.

10.4.3 Special Functions

Your application may require some combination of I/O and processing functions that is best performed by a specialized device with appropriate hardware, including a microprocessor. Such applications can usually be performed by general purpose hardware and the system CPU, but at much lower speeds. For example, the DRE-11 supports alternate buffering of input data, allowing you to avoid software buffer management operations that would be required with the DR11-W. The DR780 and DR750 VAX interfaces will fetch commands from a list and perform the requested operations in microcode without interrupting the VAX processor. Matching these kinds of features to your own specific application requirements is the best way to ensure adequate performance.

10.4.4 Buffer Memory

Buffer memory in the I/O interface is a special feature with very wide applicability to high performance realtime tasks. Such memory, usually in the form of first-in/first-out (FIFO) buffers, often makes the crucial difference in sustaining high-speed input operations. The FIFO is used temporarily to capture and store incoming data, while the system bus or CPU is unavailable to service the operation. Without buffering, it is often impossible to sustain maximum data rates for more than short bursts. Options that feature FIFO buffering include the DR780 and DR750.

Chapter 1
Bibliography

1.0.1 Bibliography

1. Artwick, B. A. *Microcomputer Interfacing*. Englewood Cliffs: Prentice-Hall, 1980.

   Comprehensive treatment of analog and digital interfacing topics. (Intermediate)

2. Benedict, R. P. *Fundamentals of Temperature, Pressure, and Flow Measurements*. New York, Wiley, 1984.

   Theory and practice of transducer design. (Advanced)

3. Garrett, P. H. *Analog Systems for Microprocessors and Minicomputers*. Reston: Reston, 1978.

   Transduction, conditioning, acquisition, and transmission of analog signals. (Advanced)

4. IEEE Standard 488-1978. *Digital Interface for Programmable Instrumentation*. New York: IEEE, 1978.

Comprehensive description of the General-Purpose Instrument Bus (GPIB). For copies, contact The IEEE, Inc., 345 East 47th St., New York, NY. (Intermediate)

5. Loriferne, B. *Analog-Digital and Digital-Analog Conversion.* London: Heyden, 1982.

   Principles and design of analog interfaces. (Advanced)

6. McNamara, J. E. *Technical Aspects of Data Communication.* Bedford: Digital Press, 1982.

   Synchronous and asynchronous serial data communication. (Intermediate)

7. Mellichamp, D., ed., *Real-Time Computing - With Applications to Data Acquisition and Control.* New York: Van Nostrand Reinhold, 1983.

   Comprehensive overview of realtime computing. (Intermediate)

8. Morrison, R. *Grounding and Shielding Techniques in Instrumentation.* New York: Wiley, 1977.

   Comprehensive treatment of noise and shielding in electrical systems. (Advanced)

9. Sheingold, D. H. *Transducer Interfacing Handbook.* Norwood: Analog Devices, 1980.

   Temperature, force, pressure, flow, and level transducers - theory and applications. (Intermediate/Advanced)

10. Sheingold, D. H. *Analog-Digital Conversion Notes.* Norwood: Analog Devices, 1980.

    Theory and applications of A/D converters. (Intermediate)

11. Taylor, J. L. *Computer-Based Data Acquisition Systems - Design Techniques.* Research Triangle Park: Instrument Society of America, 1986.

    Measurement error and sampling theory in data acquisition systems. (Advanced)

Index

F

G

H

I

J

L

M

N

O

P

R

S

T

V

EB-29499-61

Digital Equipment Corporation • Marlboro, MA 01752