AN ABSTRACT OF THE THESIS OF

Clayton R. Stanford for the degree of Master of Science in Electrical and Computer Engineering presented on December 6, 1996. Title: Guidelines for Implementing Real-Time Process Control Using the PC.

Abstract approved: Redacted for Privacy

James H. Herzog

The application of the personal computer in the area of real-time process control is investigated. Background information is provided regarding factory automation and process control. The current use of the PC in the factory for data acquisition is presented along with an explanation of the advantages and disadvantages associated with extending the use of the PC to real-time process control. The use of interrupt-driven and polled I/O to obtain real-time response is investigated and contrasted with the use of a real-time operating system. A unique compilation of information provides guidelines for selecting an implementation method for real-time control. Experimental work is performed to evaluate the access time and latency periods for the hard drive, video monitor, and I/O devices operating in a DOS environment. The execution speeds of C and assembly language programs are investigated. A method to estimate the performance of a real-time control system using polled or interrupt-driven I/O is developed.

Guidelines for Implementing Real-Time Process Control Using the PC

by

Clayton R. Stanford

A THESIS

submitted to

Oregon State University

in partial fulfillment of
the requirements for the
degree of

Master of Science

Presented December 6, 1996
Commencement June 1997

Master of Science thesis of Clayton R. Stanford presented on December 6, 1996

APPROVED:

Redacted for Privacy

Major Professor, representing Electrical and Computer Engineering

Redacted for Privacy

Head of Department of Electrical and Computer Engineering

Redacted for Privacy

Dean of Graduate School

I understand that my thesis will become part of the permanent collection of Oregon State University libraries. My signature below authorizes release of my thesis to any reader upon request.

Redacted for Privacy

Clayton R. Stanford, Author

ACKNOWLEDGMENTS

I wish to thank my wife, Aninha Soule Brill, for her enduring support of my work on this research project. Without her guidance and motivation, this work would not have been possible. My children, Evan and Maya, have also been very understanding and patient during the long hours of research and editing in which I was unable to be with them. My family deserves the merits of this thesis as much as I.

Professor James Herzog has provided excellent technical and editorial input on this thesis. His support and assistance were key to the successful completion of this effort. Jim provides the students of Oregon State University with a balance of theoretical and practical knowledge in electrical engineering, which is integral to success in the real world.

TABLE OF CONTENTS

TABLE OF CONTENTS (Continued)

TABLE OF CONTENTS (Continued)

LIST OF FIGURES

LIST OF FIGURES (Continued)

GUIDELINES FOR IMPLEMENTING REAL-TIME PROCESS CONTROL USING THE PC

CHAPTER 1
INTRODUCTION

1.1 Background

The concept of *process control* (Figure 1.1) appears in every discipline of engineering. In many applications it is mandatory that the controller respond to external events within a limited time determined by the dynamics of the process. This type of controller is termed a *real-time controller*. When the response time of the controller must be minimized to keep pace with a high-speed process, stringent requirements are placed on the computer system implementing the control function.

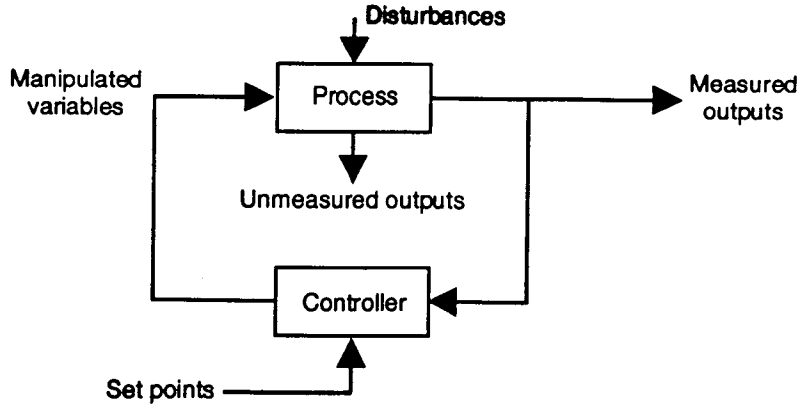

Figure 1.1. Process control.

Factory automation techniques make extensive use of real-time control [1-4]. The conventional approach for implementing a control system within a factory uses a hierarchical design (Figure 1.2). At the lowest level, a programmable logic controller (PLC) provides local control for a process. The PLC consists of an embedded microprocessor with multiple control ports and a single interface for high-level communication. The PLC is required to perform real-time process control while maintaining high reliability.

The next layer of control is typically implemented using a minicomputer or workstation to provide supervisory control over the PLCs, forming a loosely-coupled system. The minicomputer is connected to the PLCs through a variety of interfaces with point-to-point communications commonly using RS-232 and network communications commonly using proprietary PLC protocols. Tasks performed at this level are more global in nature, such as orchestrating the start up and ongoing synchronization of an automated assembly line. The minicomputer must provide a reliable interface to all the processes of the production line simultaneously. This requirement of providing fault tolerant, real-time, multitasking control creates a technical and economic bottleneck that limits the application of process control to high production/high value processes.

In the most advanced automated factories a mainframe computer will reside above the minicomputers, communicating with them using standard computer networks. This computer is involved in database management, performing tasks such as trend analysis. The mainframe is not required to possess real-time capabilities, and is not involved in the local process control.

The involvement of the personal computer (where personal computer refers to the 486 and Pentium-based machine) in process control has been minimal up to this point. Less than 3% of the PCs in the factory are actually being used for process *control,* instead they are finding use in process *monitoring* [5]. An entire industry

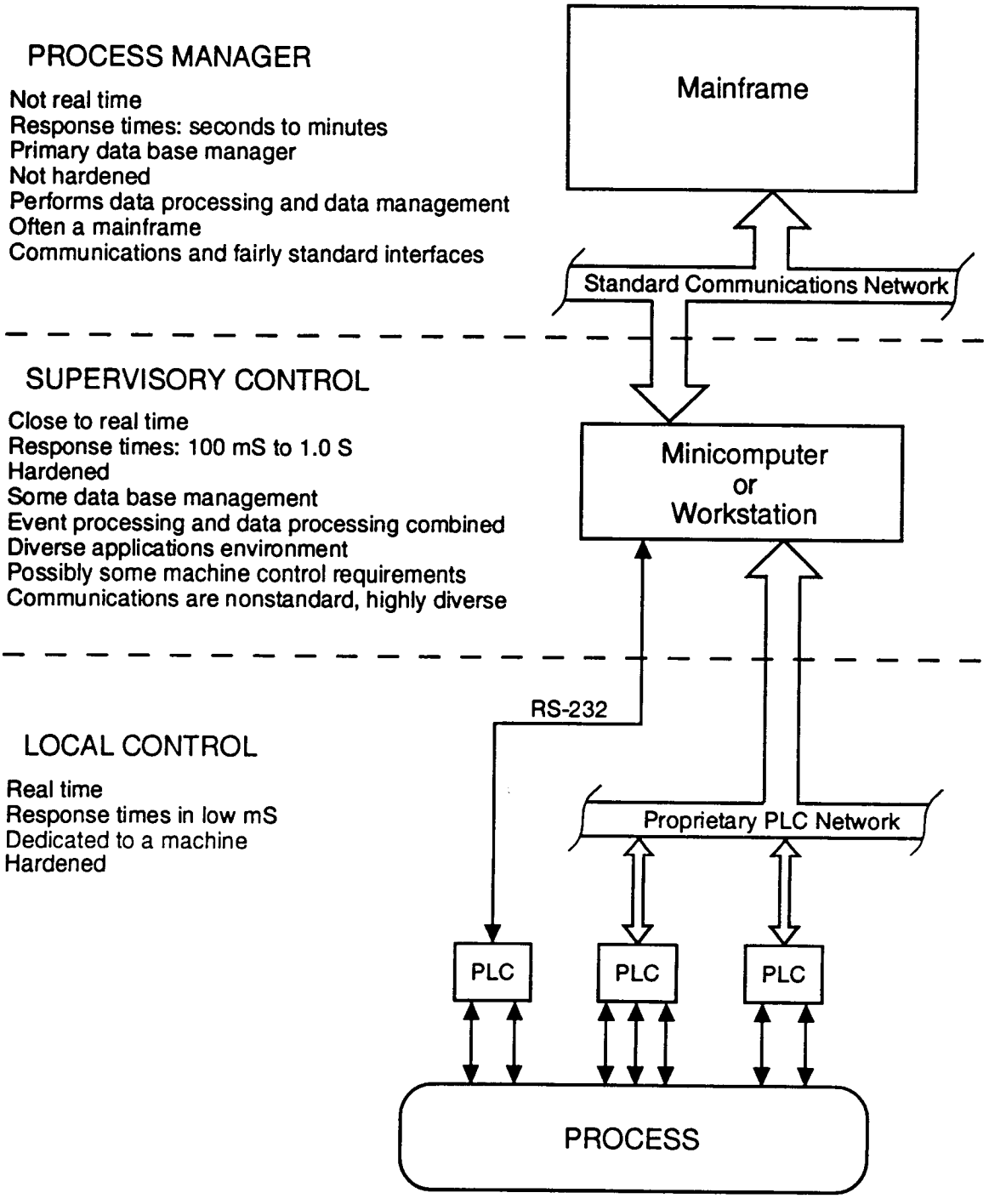

Figure 1.2. Hierarchical design of factory control systems.

supports the use of the PC for process monitoring and data acquisition by providing software and plug-in cards which enable the PC to perform a wide variety of data acquisition functions at affordable prices. The use of the PC for data acquisition has been successful, but up until recently the limited computing power of the early PCs combined with a lack of a multitasking operating system has precluded the use of the PC in advanced control applications. This is changing rapidly. GM and Ford Motor Company recently announced massive plans to replace PLCs in their factories with PC-based controllers[6]. PC-based control will permeate the control market for embedded systems and remain the dominant architecture until another desktop-computer architecture overcomes the PC's massive market foothold[7].

1.2 Objectives

The objectives of this research project are to:

- Provide background material on real-time control in factory automation, investigating the current use of the PC in industrial control and limitations in the use of the PC for real-time control.
- Provide guidelines for implementing real-time process control using the personal computer.
- Perform tests on the DOS operating system to determine the access time and latency for the hard disk, video, I/O, and clock services. (Latency created by disabled interrupts is a significant impediment to real-time control user DOS.)
- Provide a quantitative method to estimate the performance of a real-time control system using the PC.

1.3 Motivation

The motivation for this research is based on the desire to use the economic advantage of the PC to displace more expensive computer systems in control applications. A fully configured Pentium PC is selling for less than $2000 [8] as compared to workstations selling for $10,000 to $20,000. It has been estimated that reducing the cost of the computer for real-time multitasking control to less than $5000 would provide 30,000 small businesses in the U.S. with access to the process automation they desire[5].

1.4 History and Literature Review

The application of the PC for control has been an extension of using the PC for data acquisition (Figure 1.3). During the first half of the 1980s, the validity of data acquisition using the PC was established by companies such as National Instruments, Metrabyte, and Data Translation[9-11]. The first products to gain wide acceptance were plug-in modules that typically included A/D and D/A converters and several digital I/O lines. The performance of these products was only moderate, but their flexibility and price made them popular. For example, in 1984 Data Translation marketed the DT2801, a 12-bit, 13 KHz A/D converter combined with a 26 KHz D/A converter for $1195[12]. Previous to this, the least expensive method to implement computer-controlled data acquisition was through the use of a minicomputer. The combination of the PC and a data acquisition board provided engineers with an inexpensive tool for monitoring laboratory experiments and production processes. Since that time the selection of data acquisition products has grown in both performance and variety. Today A/D conversion is readily available with 18-bit resolution, sampling rates of 100 MHz, and multi-channel inputs[13-17].

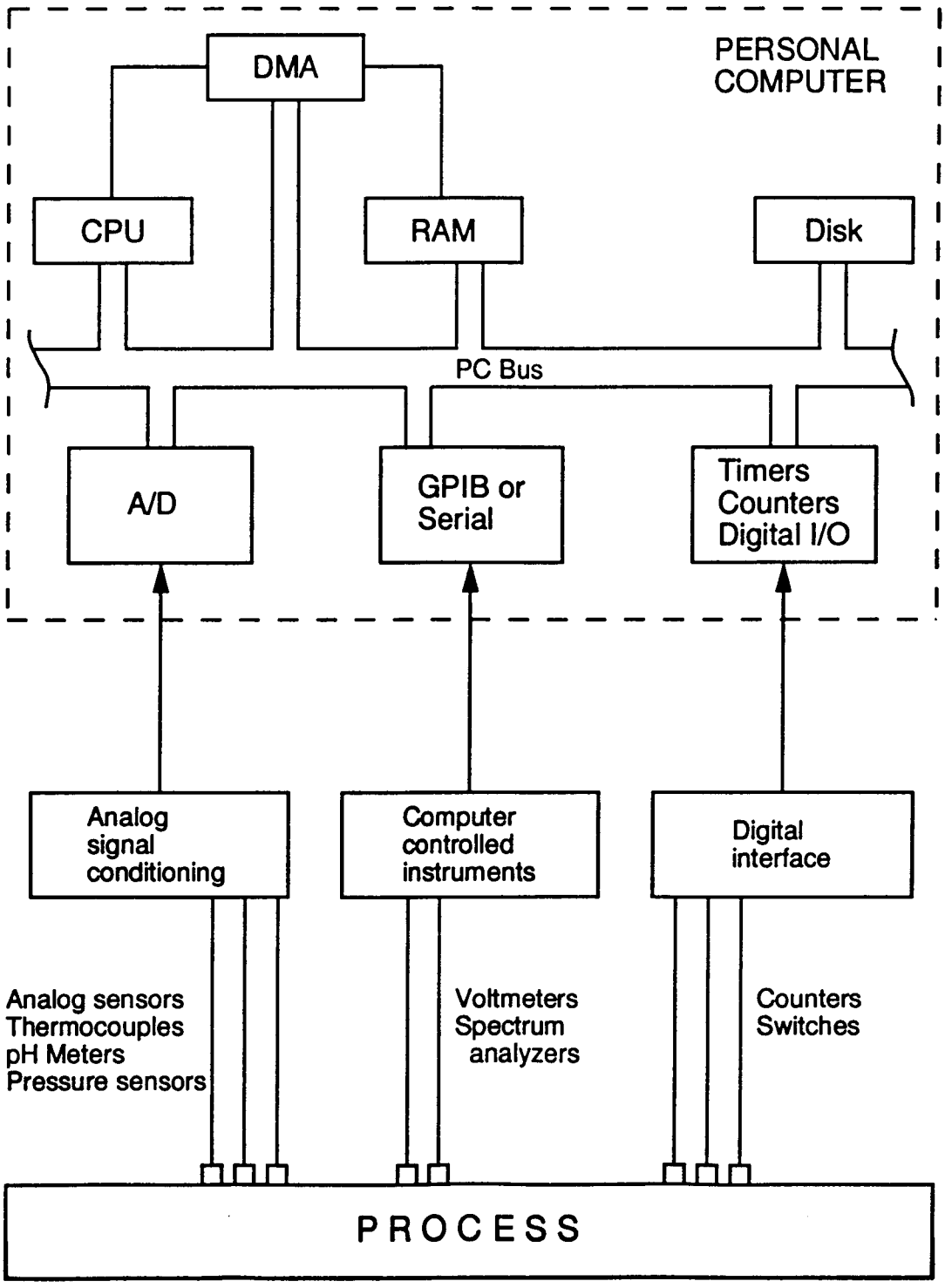

Figure 1.3. Data acquisition using the PC.

After fifteen years of development, the data acquisition industry has refined it's product offerings. Today there are plug-in modules to perform a wide variety of functions ranging from motor control to image processing. For the majority of data acquisition and control tasks, the availability of off-the-shelf modules offers a trouble-free method to provide the hardware interface between the PC and the process.

While the majority of issues involved in the physical connection of the PC have been resolved, there are several major weaknesses in PC-based data acquisition and control systems: the cost and complexity of systems integration, the inability of DOS/Windows to perform multitasking, the compatibility and stability of computer hardware, and the quality and reliability of PC platforms.

Effort is needed in the industry to provide a simplified method to implement complex data acquisition and control systems. Presently the incompatibility between different vendor's products is impeding the application of the PC for data acquisition and control[7]. There is an IEEE standard (996) for the PC/AT bus, but it only covers the bus architecture and fails to address the system level standardization needed. Systems integrators are forced to write non-reusable software or eliminate additional features in order to keep costs down. The solution to this problem will require an *open systems* approach to product design and marketing[18]. The automotive industry is probably the largest user of factory automation equipment and is leading the way toward an open architecture [7, 19-20]. The open systems approach requires standard hardware and software interfaces from all vendors. In addition, programs are implemented in a modular, object-oriented manor. Included with each application program will be a configuration program. The configuration program allows the user to tailor the general purpose program to their application by selecting a subset of the functions provided. This method allows the vendor to market one program to a wider market and simultaneously allows the user to apply the program to specialized applications.

The lack of a multitasking operating system is the primary factor limiting wide spread use of PCs for real-time control. There has been a desire for a more robust operating system since the popularity of the PC was established in the early 1980s [6, 21-22]. When multitasking systems started to appear on the market, most had evolved from home-grown code used to solve a particular problem which were then packaged to sell to the general public. These systems tended to be very limited in utilities and were only applicable to narrowly defined tasks [18]. There were two general approaches to implementing the multitasking function. To retain DOS compatibility and continue to use the large base of application programs, a shell program was written that presided over DOS. For higher performance applications, DOS compatibility was eliminated and a new multitasking operating system was created [22]. Since that time many multitasking operating systems have appeared on the market. Some of the more significant systems that have been developed over the years include *iRMX* by Intel, *OS/2* by IBM, and several flavors of UNIX [23].

iRMX by Intel was one of the earliest multitasking operating systems to appear. In addition to being multitasking, *iRMX* provided real-time response capabilities and included a complete development and execution environment. It was used for embedded control applications but had no market value outside of the industrial arena since it was not compatible with DOS. *iRMX* was both complicated to use and expensive, limiting its popularity initially. Despite these limitations, iRMX is one of the key true real-time operating systems available today. In 1991, Intel released iRMX II which is capable of coexisting with Windows 3.11[6, 24].

OS/2 by IBM received a great deal of attention when it was introduced in 1987. While most vendors working on multitasking attempted to provide a solution that was compatible with the current design of the PC, IBM redesigned the entire PC, naming the new architecture *Microchannel* and the new operating system *OS/2*. The new design was a vast improvement over the original PC and though IBM's target was geared toward the business market, the controls industry was just as excited about the new

platform [25-28]. *OS/2* and *Microchannel* together provided a solid design base for the development of the next generation of multitasking personal computers. The architecture of *Microchannel* separated the hardware from the software so that future hardware improvements could be implemented without affecting the software. Microchannel provided the first implementation of the plug and play concept, with plug in boards not requiring any jumper or switch settings. There was one major shortcoming in the *OS/2* operating system. *OS/2* could only run one DOS application at a time. This was due to IBM's decision to write *OS/2* for the 286 processor instead of the 80386 which supports the virtual 8086 mode. *OS/2* did not sell as well as was anticipated due to a combination of its inability to run multiple DOS applications, high cost, introduction of *Windows 3.0,* and the competition of other bus architectures such as VESA local bus and PCI. Today OS/2 has a small but dedicated following in the real-time control arena[29].

UNIX has been ported to the PC platform by many vendors over the years including IBM (AIX), Microsoft (Xenix), AT&T (V/386), Santa Cruz Operations (SCO UNIX) and Quantum (QNX). The merits of UNIX are well known. Some analysts predicted that PC based UNIX would become the dominant operating system as the 386/486 architecture provided enough computing power needed for these systems[30]. This never occurred principally due to the hostile user interface and the high cost of the hardware. Standard UNIX has a poor real-time response but several vendors including Santa Cruz Operations and Quantum have modified the UNIX scheduler to create real-time versions of UNIX[31-33]. These operating systems offer high quality real-time performance applicable to high-end applications.

Though not technically operating systems, Windows 3.11 and Windows 95 present the most tempting approach in which to base a control system architecture. Unfortunately, neither interface provides the response time or robustness necessary for any serious control application [34]. If one application within Windows hangs up, the entire system freezes. This lack of a recovery routine excludes Windows from use in

any high performance real-time application. Nevertheless, several vendors have used the attractive GUI of Windows as a high level executive and embedded a real-time application running under Windows [35-39]. These embedded real-time programs typically have major shortcomings when applied to advanced control applications.

Windows NT is also receiving a lot of attention from the controls industry. Though Microsoft states that NT is not a real-time operating system, many of the real-time OS vendors have NT up and running in the lab for evaluation [6]. NT has true multitasking capability and is understood to have more robust error recovery routines. Windows NT does carry a lot of operating overhead, but for slower real-time requirements, this overhead may be tolerable. An interesting point was made by Mike Gonzalez, president of Wonderware [7]. Gonzalez speculated that in two years a PC with a desktop operating system may be able to outperform the currently available PLCs with respect to response time and deterministic behavior. If this occurs, it would then be a rather straight-forward task to use the PC for control applications.

From the hardware end, the most important changes for control applications have to do with I/O rates currently being limited by the system bus. Up until 1987, the ISA bus was the only bus standard. The ISA bus is incredibly slow, yet has endured an amazing number of years. With the arrival of 386 based machines, it was necessary to expand that 16-bit bus to 32 bits. Nine clone vendors joined together to create a backward-compatible bus that supported the 80386 and also solved other limitations of the PC-AT bus [40-42]. The Extended Industry Standard Architecture (EISA) was the result. When introduced, EISA appeared that it would play a significant role in the development of the PC for control applications. The most significant improvement effecting data acquisition and control applications was the increase in the DMA transfer rate from 0.8 to 33 Mbytes/second. As is common in the PC industry, EISA has received competition from other bus architectures and has never attained significant popularity.

The most recent bus architecture to hit the PC market, PCI appears to be making a significant improvement in the archaic data rate of the ISA bus [43]. With data rates of 132 Mbytes/sec, the PCI bus will allow transfer of data from I/O cards into system memory without the need for on-card storage facilities. This improves the data capture performance of I/O cards while also reducing the price.

The latest interface method, PC-MCIA was first introduced into the notebook market. This interface has received good market acceptance and it is estimated that by 1998, 70% of desktop PCs will contain the PC-MCIA interface [44]. PC-MCIA offers the option of a small, low power interface which can be valuable for embedded PC applications.

The use of embedded PCs is expanding rapidly. This solves one of the earlier physical constrains of attempting to install industrial PCs into harsh environments. Applications that were once the domain of the PLC are quickly being invaded by embedded PCs [45]. Diskless DOS and packaging standards like the PC/104 bus are helping engineers apply the power of the PC into embedded control applications. The PC/104 bus is a miniature 104 pin stackable bus architecture coming on strong as a standard method to package the PC for small harsh applications[45]. Diskless versions of DOS provide PC compatible operating systems for use in industrial environments previously reserved for PLCs[46].

Today the hardware portion of a PC-based control system can be purchased for under $5000. A systems integrator can buy a Pentium PC developed for the personal computer market that has enough computing power for almost any control application. Plug-in cards can be purchased off-the-shelf from the data acquisition industry to interface the PC to most any process. But in comparison, the performance of software packages for control is lacking [21, 33, 48]. The systems integrator is faced with relying on software from a selection of relatively immature real-time operating system products, or developing custom programs; an expensive proposition.

CHAPTER 2
REAL-TIME SYSTEMS

2.1 Purpose

The purpose for using a real-time control system is to provide the ability to interact with the real world in a predictable and bounded manner. Digital control systems are characterized by feedback loops in which states of the process are sampled and fed to the control computer for processing. The computer then calculates the control signal to input into the process in order to obtain the desired output. The delay between the sampling of the state and the output of the control signal has a direct effect on the control characteristics. If that delay is unpredictable, then the process control is also unpredictable.

As an example of the importance of predictable and bounded response time, one may consider a computer controlling two tasks simultaneously. Task 1 requires servicing every 1.0 +/- 0.1 second. The control computer completes service of task 1 at t = 0, and then waits in a loop until task 1 or 2 needs servicing. At t=.9 seconds, an interrupt is asserted requesting service of task 1. If the computer is busy servicing task 2 and does not get to task 1 within .2 second, the timing requirement is violated. This failure may result in a degradation or destruction of the materials involved in the process.

A time-sharing system such as UNIX is unacceptable for real-time control because there is no provision to preempt system calls[32]. If the second task in the example above executes an extended system call such as file creation, interrupts are disabled while the system call is being executed. This type of instruction can take

several seconds to complete, and in the example above would cause a failure in the timing of task 1. It is only with a preemptive kernel that a predictable response time can be assured.

2.2 Response Time Classification

Real-Time systems are driven by the external world which imposes demands on the controller to meet time deadlines. The controller can be given a speed classification based on these response time constraints[3].

Low-speed applications are characterized by a control system that does not need to hurry in order to service the process. Inefficient programming techniques suffice to meet process deadlines which are in the range of seconds to minutes. Room temperature control is an example of a low speed application.

Medium-speed applications require some optimization of program execution in order to keep up with the process. The computer is kept busy servicing the process and prioritizing of tasks may be required. Response times are in the milliseconds to seconds range.

High-speed applications require response times nearly equal to the capacity of the computer. The program must be completely optimized and servicing of the process is often very simple out of necessity. Response times in the microseconds to milliseconds range are common. Applications include digital signal processing and servo loops.

The response time classifications above are relative to the throughput of the controlling computer. This indicates that the response time of a controller can be improved by using a faster computer to execute the control function. This is a valid

assumption to an extent, but I/O often limits controller throughput. Figure 2.1 illustrates a typical interface between a process and a control computer in which the I/O is the limiting factor determining throughput. In this example the 1200 baud I/O link passes one 16-bit word every 17 mS, not including idle time. If the computer can calculate the new value to send to the input before the next output sample arrives, then the controller response time will be limited by the data link and not the computer. Therefore both the CPU throughput and I/O bandwidth are important in high-speed applications.

The speed classifications above are also dependent on the complexity of the control algorithm. For example, in Figure 2.2 the controller samples the error signal and then calculates an appropriate control signal to drive the output to the desired set

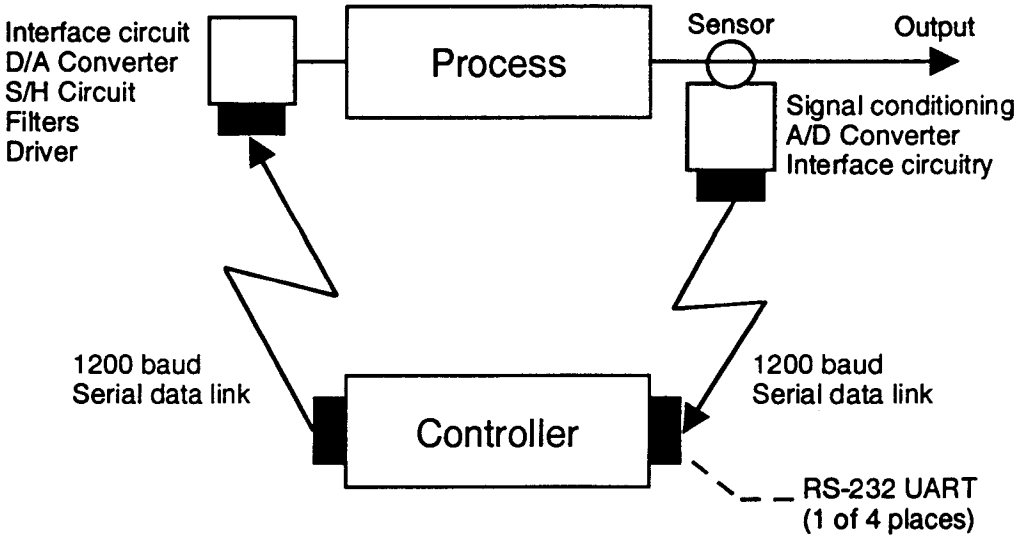

Figure 2.1. Typical RS-232 interface between the computer and the process.

point. The rate at which the samples must be taken is dependent on the process time constant. A common method to derive the control signal from the error signal is proportional-integral-derivative (PID) control in which the control signal is defined as:

$$\text{control signal} = K_p * \text{error} + K_i * \int \text{error}\, dt + K_d * d(\text{error})/dt.$$

The constant K_p controls the proportional feedback, K_i controls the integral feedback, and K_d the differential feedback. This method minimizes the error but requires a minimum of three multiplications, three additions, and one subtraction[49].

For applications in which the process time constant is short, it may be difficult to perform the computations within the allocated sample period. In contrast the simplest

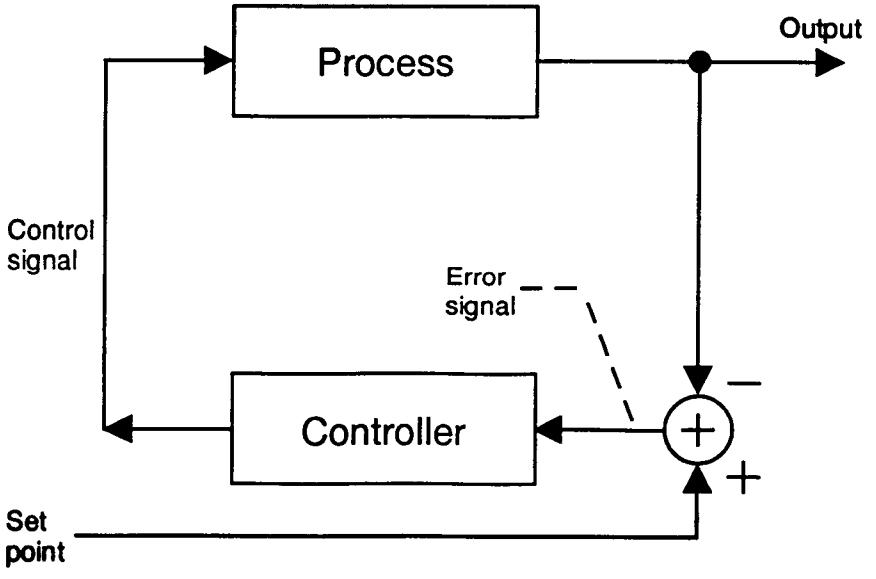

Figure 2.2. Signals associated with process control.

control algorithm, on-off control, determines the control signal by executing the following code:

```
if ( error > 0 )
control_signal = K;
else
control_signal = 0;
```

Thus in this example, the computation has been reduced to one conditional branch in exchange for less sophisticated control. Therefore it may be possible to ease the speed requirement of the system by reducing the control algorithm complexity.

2.3 Performance Measures

To quantify the performance of a real-time system, the following three performance measures are used:

Interrupt-response time or interrupt latency: The time required from the receipt of an interrupt until the interrupt service routine (ISR) is invoked is called the interrupt-response time. This includes the time to save the program counter, vector to the ISR, and begin execution. The worst-case value would include the longest period in which interrupts are disabled.

Context-switching or Task-switching time: The time required to switch between two tasks in a multitasking operating system is called the context switching time. This is the overhead time associated with the time multiplexing of several tasks and includes the time to save the state of the current task, locate the new task, and load the new task. The worst-case time must include the longest period in which interrupts are disabled.

Task-response time: The time spent from receipt of a hardware interrupt until the operating system dispatches the high priority task requested is called the task-response time. Task-response time includes the interrupt-response time, the time to process the ISR, the time to evaluate the priority level, and the task-switching time. This is the most important measure of a real-time system. The task-response time must be predictable to insure the integrity of a real-time system.

These performance measures provide a quantitative method to evaluate real-time operating systems. Each vendor may use slightly different definitions of these indices; therefore, when evaluating an operating system it is important to understand the exact definition used. In particular, close attention must be paid to average values versus worst-case values, where the worst-case values are typically of greatest significance. It is also important to understand that for some systems, the latencies are a function of the number of tasks being executed. Evaluating real-time operating systems is a difficult and time consuming task [50]. Performing a thorough evaluation for a particular application may require experimentation with sample programs.

2.4 Architecture

Many real-time applications use standard computer hardware with the specialization incorporated into the software. Nevertheless, certain architectural forms are desirable.

The most important hardware attribute of a real-time computer is a flexible, multilevel interrupt structure since interrupts are the main method of interfacing the CPU to the process. The interrupt structure is usually implemented using a programmable interrupt controller or PIC (Figure 2.3). The PIC accepts multiple interrupts from the I/O devices and processes the interrupts according to a priority rating before allowing an interrupt to divert execution of the CPU. The lower priority

interrupts are held in a queue for processing if the CPU is busy with a higher priority interrupt. The programmable nature of the PIC also allows the user to mask off interrupts and rearrange the priority levels. In conventional computers there are usually eight to sixteen interrupt lines entering the PIC. If the computer must interface with more devices than there are interrupt lines, the interrupts are chained to allow two devices to share one interrupt line. During the interrupt service routine each device is polled to determine which device requested servicing. Chained interrupts are undesirable in real-time systems due to the added time required to service the interrupts and the variability in the interrupt-response time. It is therefore desirable to have individual interrupt lines for each device that the computer will control. This can be accomplished by incorporating multiple PICs in a hierarchical configuration.

Memory protection is used to provide security against accidental corruption of data or program code in a multitasking environment. The protection consists of hardware that prohibits a task from writing to any memory outside of its allocated address space. With this hardware protection, an error in one software module can not

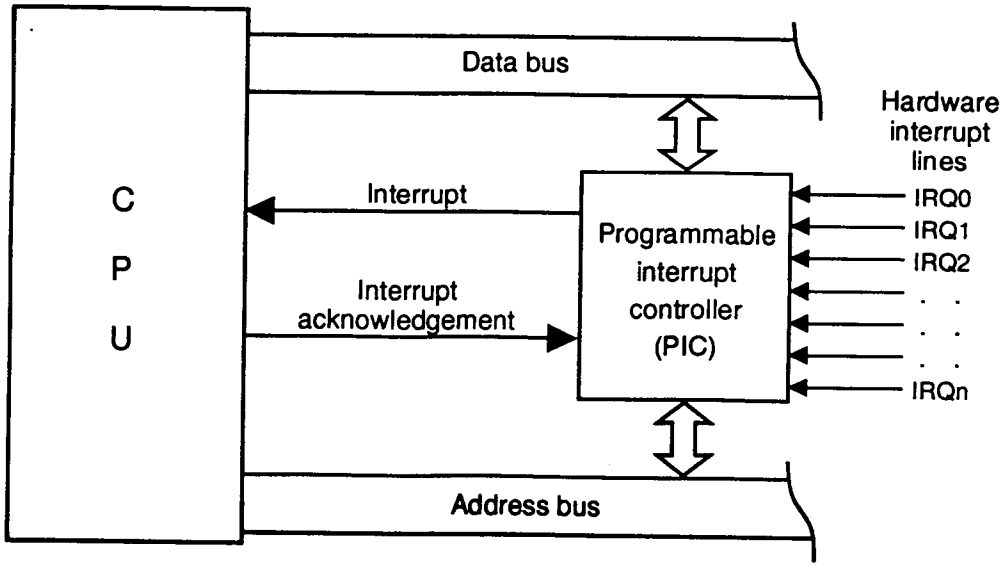

Figure 2.3. Interrupt structure using programmable interrupt controller.

damage the operating system or any other task in the computer. This improves system integrity by eliminating a potential cause of a system crash and helps to contain errors within a restricted area.

A real-time clock is used in process control to schedule events at fixed times. There are two common methods to schedule events: either the clock is programmed with the desired time and supplies an interrupt to the CPU to implement the task, or the clock is used only to supply the current time and event scheduling is handled by the CPU directly. The first method unloads some of the real-time task scheduling from the CPU and places the responsibility on the clock, but requires a more complex clock. The second method forces the operating system to monitor time in case an event requires initiation.

2.5 Fault Tolerance

The controllers just discussed are often involved in governing processes of significant value or processes that operate with hazardous materials. For this reason the issue of fault tolerance is of concern to the control engineer[51]. In any computer system hardware or software failures can result in an abrupt end to the computer operation. Real-Time systems have an added failure mechanism if they do not meet the time deadlines imposed by the process time constant. In this case the result will be a degradation or loss of control.

To increase the reliability of control systems, hardware and software redundancy is employed. Hardware redundancy uses multiple computer systems operating in parallel with a voting mechanism at the output. Software redundancy is implemented by independently developing two or three control programs and then executing them in parallel and voting on the output. As long as the failures are independent, hardware and software redundancy can improve the fault tolerance of the system.

Transients such as power line noise or EMI can cause correlated failures for which parallel redundancy is unable to provide protection. Fault tolerance in the event of correlated failures requires the use of time redundancy. In this type of fault tolerance the system design allows for the transient event to occur, and then implements a recovery algorithm to bring the system back under control. For time redundancy to work the system design must allow for slack time so that the controller can regain control.

CHAPTER 3
THE PERSONAL COMPUTER IN REAL-TIME CONTROL

3.1 Advantages

The use of the personal computer for process control is worthy of consideration because of the economic benefits it can provide. The high volume production of the personal computer along with intense competition between clone vendors promotes technological innovation while simultaneously driving prices down. Today the performance gap between the PC and the workstation has narrowed to the extent that the distinction between the two is difficult to discern (Figure 3.1) [52]. At the same time the price-performance ratio of the PC and associated peripherals is superior to that of the workstation, making the use of the PC in new applications very attractive (Figure 3.2) [53]. When considering the personal computer for real-time control applications, there are definite advantages the PC can offer:

Inexpensive computing power: The availability of inexpensive computing power can be used as a new means to solve difficult control problems. In the past it was necessary for the engineer to spend a considerable amount of time optimizing code in the critical paths of a high-speed control application. As software costs increase, the engineer may find it economical to buy a more powerful computer to solve speed problems. In the future, an inexpensive fault-tolerant computer system might be created using three PCs with a voting mechanism. The economical PC would be particularly advantageous in this application since hardware redundancy tends to drive the system price up but would not be a big factor in a PC-based system.

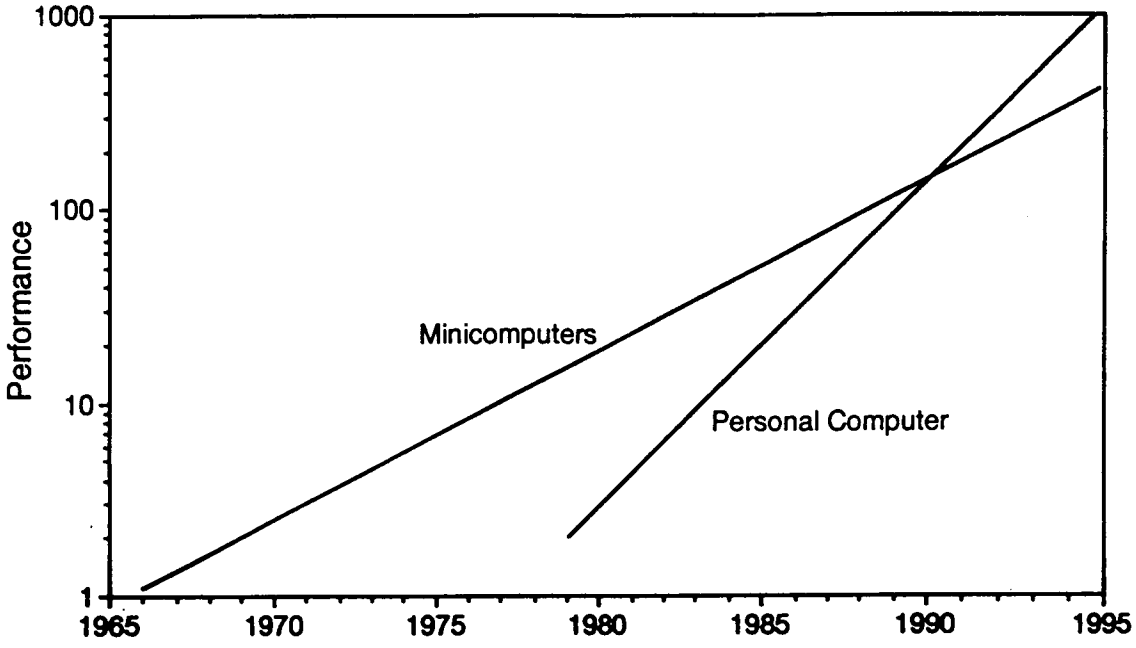

Figure 3.1. Relative performances of PC and minicomputer.

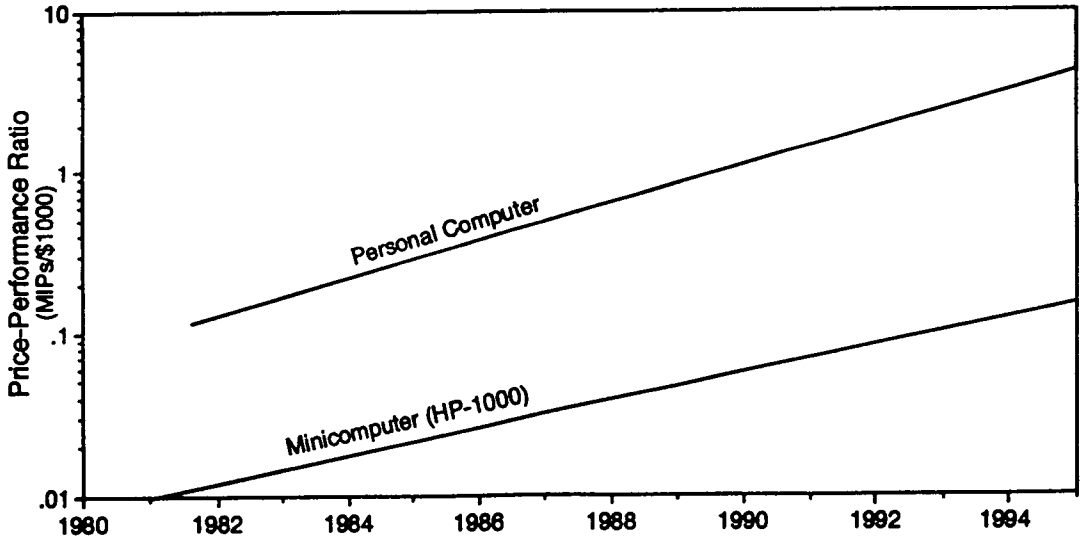

Figure 3.2. Price-Performance ratio of PC & minicomputer.

Maintainability: Whether or not fault-tolerant techniques are employed, a goal of all systems is to minimize down time, and in this regard a system based on the PC has the advantage of easy maintenance. In the case of a computer failure, the modular design of the PC combined with its prevalence in the work place makes it a simple matter for a technician to swap components in order to get the main system up and running quickly. This can lead to significant time and cost savings as compared to having to maintain a service contract with the minicomputer manufacturer.

Software development costs: The popularity of the PC provides the benefit that software costs are reduced due to mass marketing. As the sophistication of control systems has increased, the software costs are becoming the major expense. The PC offers the following software advantages: inexpensive development tools, a large base of application programs available, the convenience of being able to develop applications on the target machine, and a greater availability of computer programmers for the PC as compared to other platforms.

Data acquisition interface modules: The other major advantage to using the PC comes from the development work already performed by the data acquisition industry which offers interface cards for virtually any application. These include motion control cards, A/D and D/A boards, and communication cards for GPIB, *Ethernet*, and RS-232 protocols.

3.2 Disadvantages

The personal computer also has some disadvantages that affect its applicability to control tasks:

DOS operating system: The single-tasking operating system of the personal computer creates the biggest obstacle in the application of the PC for control. While the

hardware portion of the PC has steadily increased in performance, little change has occurred in DOS. The 286 microprocessor, which incorporated memory protection to facilitate multitasking was implemented into the IBM AT in 1984, yet only recently has significant use of multitasking occurred.

Nonreentrant code: Many of the service routines within DOS contain segments of nonreentrant code. A service routine that is nonreentrant stores variables in memory instead of on the stack or in registers. If the service routine is called again while the current routine is active, the second routine will become nested within the first. Due to this method of storing variables, the context of the first routine will be overwritten by the second, usually resulting in a system crash. (For a more detailed discussion of reentrancy refer to [22, 54-56].) As a single-tasking operating system this was not a significant problem. Now that there is interest in multitasking, the non-reentrancy of DOS creates a significant functional limitation. To avoid reentrancy conflicts, interrupts must be disabled during system calls. This can result in long, unpredictable periods of latency before an external interrupt is serviced, thereby violating one of the basic requirements of a real-time system.

Operating system support: The PC is arranged with the lowest level of programming services stored in ROM-BIOS and the next higher level incorporated into DOS (Figure 3.3). Operating system services should be designed to support the applications programmer by providing a buffer to separate the hardware from the program for portability. The services that were provided in the original PC were inadequate to support hardware such as the video display and the serial port. This has caused programmers to write directly to the hardware in order to achieve higher performance. Having to support the hardware within application programs adds a significant burden to the programmer's job, and consequently increases development costs. Windows 3.11 and 95 have attempted to solve this problem, but at the cost of added complexity and slower speed.

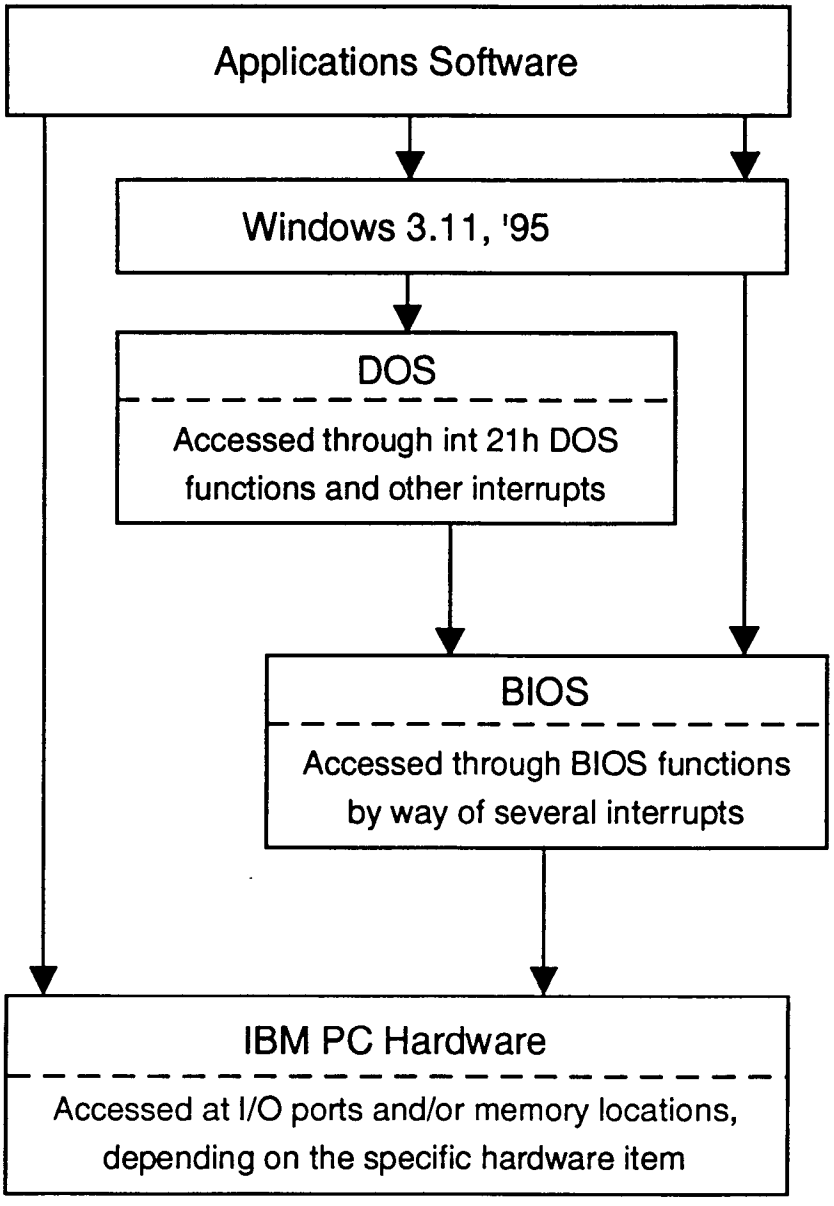

Figure 3.3. PC hardware/software architecture.

System integrity: The PC is assembled from subsystems manufactured by many different vendors. Considering that there are few specifications defining the hardware and software interfaces, these subsystems fit together surprisingly well. For high reliability applications, the lack of a tightly integrated system creates certain risks. The typical user of a PC in an office environment can select software and hardware from a wide range of sources and combine them together with a good probability that the system will work. If an interface problem occurs, the user can find a different combination of resources to perform the task. In process control more assurance is needed that the system will perform without errors. In contrast to the PC, companies like Hewlett Packard have an advantage in that they control all aspects of their minicomputer development from architectural design to the coding of the operating system. This provides them with more confidence that their system will work without failures.

Number of interrupts: A basic hardware limitation facing the PC when applied to process control is a lack of interrupts. The ISA bus has eleven edge-triggered interrupt lines which EISA modified to be configurable as edge- or level- triggered. An advantage of level triggering is that multiple sources can use a single interrupt line by tying all the interrupts together, creating a logical OR function. This is an improvement, but still creates limitations since it becomes necessary to poll each possible interrupt source to find the requesting unit, causing a delay in servicing.

Radiated EMI: The last hardware issue of importance is regarding the noise EMI problem of the PC. Microprocessor systems are notorious for radiating EMI. The original PC design did not properly address radiated noise in the physical configuration of the bus. Strict EMI requirements such as the European CE mark[57] will help address the environment external to the PC, but do nothing for the environment internal to the PC chassis. Low level signals that are processed using plug-in modules inside the PC are susceptible to corruption due to the high levels of EMI within the chassis. It is common to see Faraday shields mounted on plug-in boards to protect sensitive signals from corruption. Another method to solve this problem is to provide external

amplification of the signals to raise the noise level above the internal noise of the PC. Either method adds cost, complexity, and in some cases compromises system performance.

CHAPTER 4
SELECTING AN IMPLEMENTATION METHOD FOR REAL-TIME CONTROL

4.1 Design Approach

The design approach for a control system must start with an analysis of the process needs and end with the selection of a computer system. A common mistake is to specify a hardware platform without giving careful consideration to the software details necessary to complete the system. The following guide may be used to determine the best approach to a particular control problem.

Define the process and operator needs: This should be the first objective. Consult with the process engineers and the plant floor operators to determine the important variables and controls in the process.

Select a control architecture: The design should address the process interface as a high priority. In implementing the control hierarchy of Figure 1.1, consider that the number of levels of control can vary from one to six or more. If the control system is implemented with multiple layers, the design will provide for a graceful degradation in performance if there is equipment failure, but the communication needs will be more involved. A tradeoff between the extent to which the supervisory computer is involved in the control function of each node and the number of nodes the supervisor oversees must be made. It is usually best to balance the loads on a controller so each task requires a similar amount of computer involvement and time. All control options including manual control, smart sensors, PLCs, single-board computers, specialty controllers, plug-in boards, coprocessors, PCs, and workstations should be kept in

mind when selecting an architectural form. (Refer to [58] for an overview of the various controllers available.)

Select control hardware: The selection should be based principally on the availability of software that will perform the required functions. Writing custom software has become very expensive and should be avoided. Unless the application is very simple, highly specialized, or will be installed in many locations, the use of commercial software is important in order to keep costs down. The second consideration in hardware selection should be an analysis of the communication requirements of the project. The lower layers of the hierarchy, where real-time response is critical require a predictive communication network such as a token passing network like IEEE 802.4 [59]. This type of network, though possibly slower than a network such as Ethernet, provides a known worst case response time. In addition, factory control systems need to be flexible and expandable. It is important to have a method of expansion available with either reserved computing power or a method to add more processors as necessary.

4.2 Applying the PC to Real-Time Control

During the design phase, consideration of the personal computer for a particular application will need to be addressed. When selecting the PC as a candidate, the following guidelines will help evaluate whether the PC is an appropriate choice.

4.2.1 Basic Limitations

Before discussing general guidelines for using the personal computer in process control, it is necessary to address some applications for which the PC is not suited.

Life critical applications: The reliability of the PC does not warrant use in any application where endangerment to life could occur. In any control application it is important to ensure that a hazardous condition does not develop in the event of a computer failure.

High-reliability processes: The personal computer can not be depended on for the more demanding fault-tolerant applications in which a failure in the control system would be disastrous to the control of a process.

Hostile environments: The packaging of the conventional PC precludes operation in unprotected factory environments. For more demanding applications an industrial PC can be used, but these units tend not to differ significantly from consumer PCs. (Refer to [60] for a detailed discussion of hardened personal computers.) If the local process I/O is handled by a PLC which is hardened, the PC can be located in a protected environment a distance from the process.

Technically demanding applications: If the best technology available in the controls industry to perform the desired process control is marginal or insufficient, then the PC should not be considered for the application. The ability of the PC to perform control functions is only average as compared to specialized controllers, therefore the PC is not a good choice when technological limits are being pushed.

4.2.2 Local Control

The PC has the ability to fit into the control hierarchy at several levels, from local cell controller to higher level supervisory positions. There may be one or more places in which the PC can provide a viable solution. The following guidelines may be utilized to assess whether the PC is appropriate for local process control.

The PC may be employed for local control of process variables by interfacing the PC to the process with plug-in modules from the data acquisition industry. This approach provides the least expensive solution to simple control problems. The interface boards typically have low point counts and rely on the PC to provide each operation. The following issues will need to be addressed:

Number of controllable points: The number of points a PC can control is dependent on the time constant of the process, the amount of computation required, the data rate and resolution of the interface, the computer throughput, and the control method selected. For a high-speed application, a single variable will be all that can be controlled. For a low-speed application such as temperature control, perhaps up to twenty points may be controlled. A method to estimate the number of points a PC can control is presented in chapter 5.

Hardware/software integration: Difficulties in integrating hardware and software subsystems is a leading cause of problems in the implementation of a control system. Careful attention must be paid to the compatibility between the products of various vendors. In particular, the requirements for integrating device drivers into the selected application program should be noted; writing new device drivers can be expensive.

Sensor interfacing: The sensor signals in a local control configuration are brought into the PC chassis for processing. If any of the signals contain low-level analog information it is necessary to take precautions to insure that these signals are not corrupted by radiated EMI from the PC. Signal shielding will be required and in the more sensitive applications external preprocessing may be necessary. The preprocessing would consist of amplification to raise the noise level of the signal above that in the PC, or performing the A/D conversion externally. In the case of external A/D conversion, a sharp reduction in sampling rate may occur if DMA is not employed. In either case the added expense will be significant due to the external hardware.

Environmental considerations: It is generally necessary for the PC to be located close to the process to avoid the expense of running long sensor wires. This means that a protected area must exist for the PC or the use of an industrial PC is required.

When the use of the PC for direct control creates a situation in which some aspect of the control application can not be implemented due to a limitation in performance of the PC, the addition of a PLC, dedicated controller, or a coprocessor should be considered. For example, if the control application consists of two low-speed processes and one high-speed process, by using a dedicated controller for the high-speed process the load on the PC can be balanced and it may then be possible to perform the control functions at the rate desired. The off-loading of responsibilities to local controllers leads to the use of the PC for supervisory control.

4.2.3 Supervisory Control

In Figure 1.1 the upper layer of the control hierarchy uses a supervisory controller to oversee the actions of multiple local controllers. The supervisor provides the local controllers with information like start times, set points, and process recipes, and receives information like process status and alarm conditions. While the local controllers are positioned close to the process, the supervisor is typically located some distance away and communicates with the local controllers via RS-232, GPIB, and proprietary computer networks. The characteristics of the PC make it more suitable for supervisory control than for local control. The following topics address the important issues in evaluating the use of the PC for supervisory control:

Number of controllable points: When operating as a supervisor, the PC can oversee more points than when operating as a local controller. The supervisor is relieved from the intense I/O and stringent response-time requirement associated with

local process control. Depending on the scan rate required, this allows the PC to oversee between five and thirty local controllers. Refer to chapter 5 for a method to estimate the number of local controllers the PC can supervise.

Hardware/software integration: With supervisory control, the principal interfacing is between the PC and the local controllers through plug-in communication cards. Compatibility issues between the equipment of different vendors should be anticipated. Each PLC vendor uses a different communication network commonly referred to as a data highway. The vendor will supply a plug-in card for the PC which provides the hardware link and a device driver for the software link, but a compatibility problem may occur if the control software running on the PC does not support communications on the data highway of the PLC vendor. In addition, the low-speed nature of the RS-232 serial protocol, which is the most common method to connect general purpose equipment, may limit the throughput of control functions.

Environmental considerations: In supervisory control the environmental requirements are relaxed because the computer can be located away from the process in a protected area. In most applications a standard PC can be used without the need of protective measures.

4.3 Software Structures for Control

There are three basic choices in software structure for implementing real-time control on the personal computer: polled I/O, interrupt-driven I/O, and the real-time operating system. Each method has inherent advantages and disadvantages that affect the application of the PC for process control.

4.3.1 Polled I/O

The simplest software structure for real-time control is polled I/O. This method implements a software loop which continuously interrogates (polls) each of its inputs to determine if service is required (Figure 4.1). Each input reads the status of a process to determine if service is requested. If servicing is necessary, the program provides the service before continuing the polling loop.

Application: Polled I/O is most appropriate for low-speed systems or systems in which the PC is only servicing a few points. In the case where the PC is dedicated to servicing a single point, polled I/O is very efficient. As the control complexity increases, polled I/O becomes inefficient and results in a highly variable response time. For these reasons, polled I/O is not suited for applications involving higher point counts. An exception to this may be an application in which the PC is servicing a system with multiple points wherein each point requires an equal and fixed service time.

Simplicity of implementation: Polled I/O has the advantage of being the most straight-forward control method to code. Standard programming practices can be used and polled I/O can be implemented in DOS without difficulty. It is not necessary to understand the details of the operation of the PC to work with polled I/O systems.

Overhead: Polled I/O generally makes poor use of computer resources. As the program loops around, polling each point to inquire if service is needed, CPU time is being wasted. It is usually necessary to keep the CPU under utilized in order to have the capacity to service the worst-case situation in which all points need attention simultaneously. Therefore a large amount of time is often wasted with the CPU looking for work, rather then performing a control function. It should be noted that polled I/O is an efficient method of control for one high-speed task. In the case of a high-speed application, servicing is needed nearly 100% of the time, making the overhead of polling minimal.

Variable response time: As the program goes through its polling loop, the fastest response occurs when only one point needs servicing and the slowest response occurs when all the points need servicing. The response time variation is a function of the number of points in the loop and their complexity, so that there is a tradeoff between consistent response time and the size of the loop. In most applications the

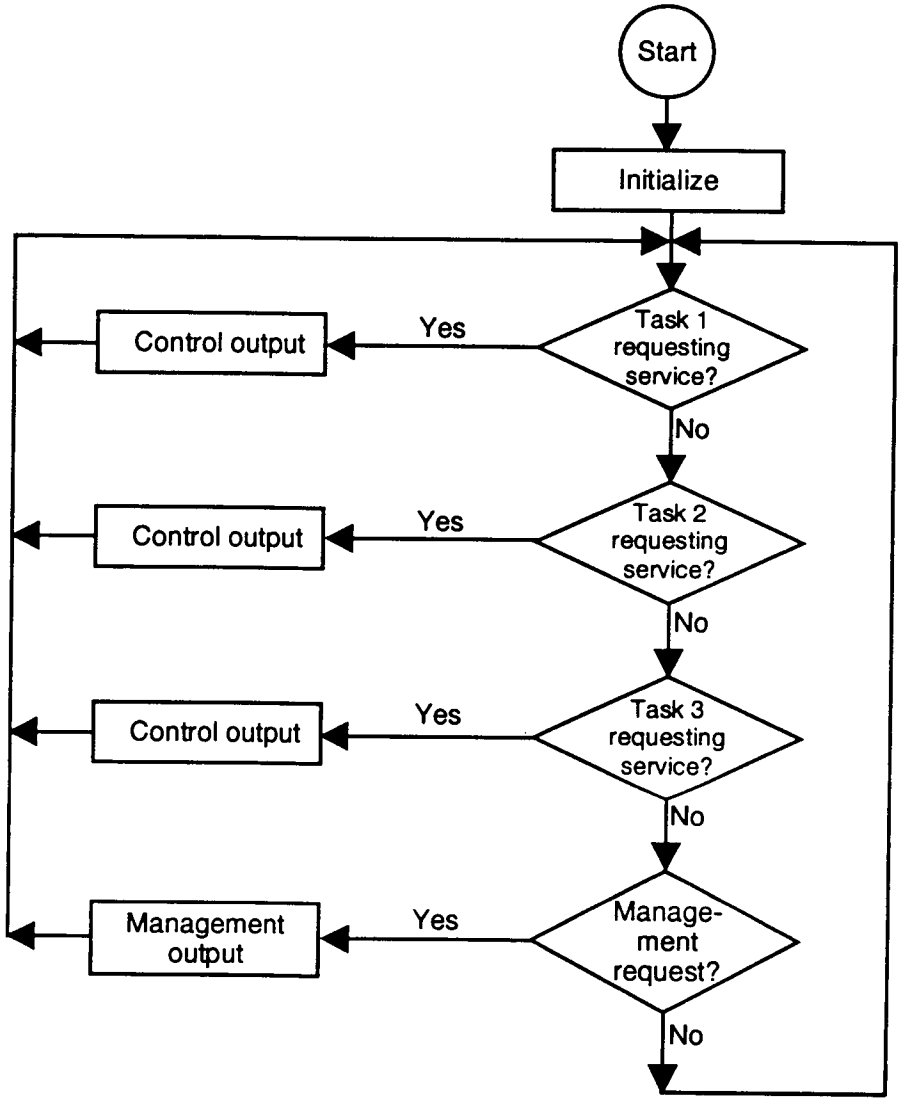

Figure 4.1. Polled I/O.

system must be designed for the worst-case response time, so the majority of the time the computer is under utilized. If an application can tolerate occasional late servicing of its control points, then the use of polled I/O without reserve computer power may be acceptable.

Lack of synchronization: The polled I/O program does not offer an efficient method to synchronize external events. This makes it unusable for applications needing real-time synchronization. In the case that internal program timing is important, such as with a sampled data system, a method to create consistent timing is required. For slower applications, use of the system clock may be acceptable. Some applications require the various loops of the program to be balanced using NOP statements to create a fixed sampling period. This method is tedious and the sample period becomes a function of the execution speed of the microprocessor.

4.3.2 Interrupt-Driven I/O

The interrupt-driven (also known as foreground/background) method is probably the most generally useful software structure. A foreground program executes a low priority program the majority of the time but is interrupted periodically by a higher priority background program (Figure 4.2). The interruption is implemented using a hardware interrupt line in conjunction with an interrupt service routine (ISR). When an interrupt occurs, control is transferred to the ISR. The ISR maintains control of the CPU until the servicing is complete and then returns control to the foreground program. Interrupt-driven I/O has the advantage that good real-time performance can be obtained from DOS. A common application of this structure uses the background program to provide real-time service to a communication channel and places the incoming data into a buffer. The foreground program operating in a non-real-time mode can then pull data off the buffer for processing. This is how most I/O is handled in conventional computer systems.

Application: Interrupt-driven I/O is suited for applications in which there are a small number of tasks that require fast servicing using a priority ranking. The tasks should not require extended CPU time and the use of disk or video services should be minimized. Interrupt-driven I/O is the most efficient method to implement real-time response because the only overhead is associated with saving the registers of the foreground program. This method is also compatible with DOS so that standard applications can run in the foreground while the ISR operates in the background.

Response time: The response time of an ISR is generally very fast, typically less then 10 uS for a modern 486 or Pentium PC. There are two exceptions that can delay servicing. If the program running in the foreground disables the interrupts, most likely during a system call, then the ISR will not be executed until after the interrupts have been reenabled. During some of the longer system calls this extends to 10 mS more. This problem degrades the performance of an otherwise excellent response time. To alleviate this problem it is necessary to control the instructions that are executed in the foreground program to insure that an extended system call does not disable the

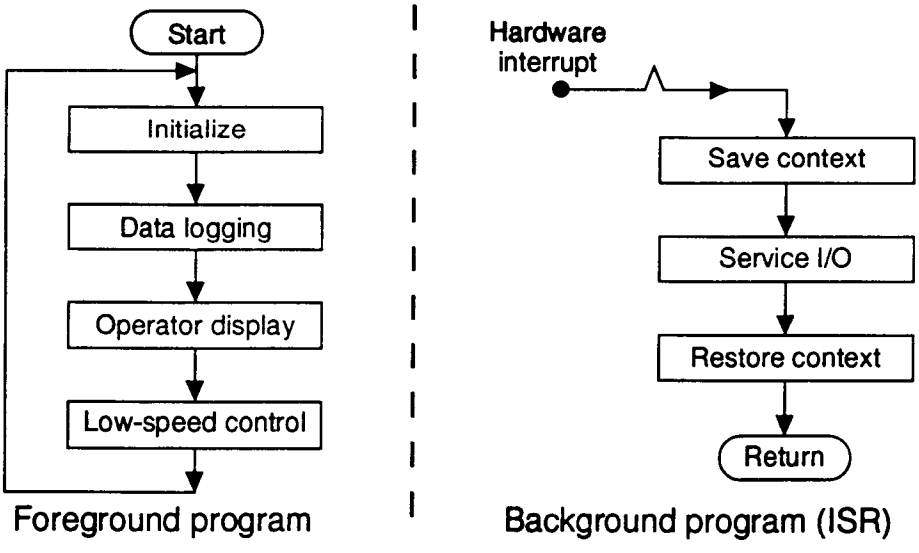

Figure 4.2. Interrupt-Driven I/O.

interrupts for an excessive period of time. Avoiding the use of system calls limits the functionality of the foreground program. It is also difficult to know when extended DOS services are being executed if the foreground program is written in a high-level language since the compiled assembly code is not visible to the programmer. The other time that servicing of an ISR can be delayed is when there is already a higher priority ISR being executed. The higher priority routine always runs to completion, thereby delaying the execution of any other program. If there are multiple ISRs, the worst-case response will occur if all the tasks request service simultaneously. For this reason it is desirable to keep ISRs as short as possible. The minimum necessary to service the interrupt should be performed in the ISR, and the remainder of the task performed in the lower priority foreground program.

Preemptive/priority execution: The structure of the interrupt system on the PC (Figure 4.3) allows an ISR to preempt CPU execution based on a priority system. As long as interrupts are enabled, a hardware interrupt can preempt program execution. If an ISR is already executing when a second hardware interrupt occurs, the higher priority interrupt receives immediate control. The PIC maintains a queue, and all ISRs will eventually execute. Interrupt 0 (IRQ0) has highest priority and IRQ8 lowest priority. A slave PIC was added to later models of the PC and the slave interrupts all have priority over IRQ3-8.

Complexity: ISRs are implemented at the lowest level of the PC architecture and require care to insure correct system operation. To implement an ISR it is necessary to thoroughly understand the microprocessor operation, the interrupt structure operation, system calls and reentrancy, and the sequence of events that occur during an interrupt in order to successfully save the context of the current program, execute the ISR, and then reinstate the preempted program. Errors made while attempting to implement an ISR will usually crash the system without leaving a clue as to the cause of the error. To add to the complexity, ISRs are usually written in assembly language for speed and hardware manipulation. For these reasons, implementing a control function

using the interrupt-driven method is more complex than the polled method and added development time should be anticipated.

Maximum number of tasks: The number of tasks that can be serviced using interrupt processing is limited. Installing more than five ISRs becomes difficult because of the complexity involved, the limit of five available interrupts on the PC, and the increase in response time in the event that all of the ISRs are requested simultaneously.

Limited ISR programming resources: It is difficult to use DOS services from within an ISR. The execution of an ISR can occur at any time and may occur while the foreground program is executing a DOS service. The fact that DOS contains

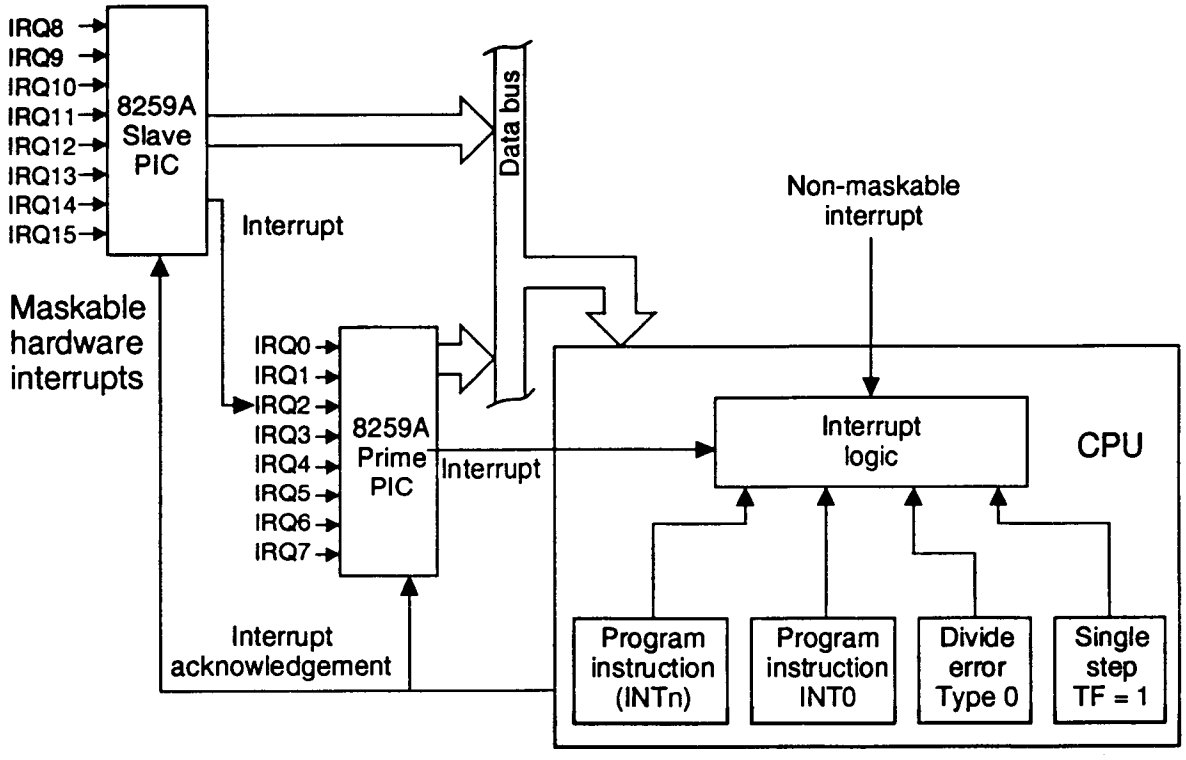

Figure 4.3. Interrupt structure of the personal computer.

nonreentrant code means that if one DOS service becomes nested inside another DOS service, the system can crash. If the ISR uses a DOS service after interrupting the execution of a DOS service by the foreground program, a reentrancy violation will occur, possibly causing the system to crash. Disk and video services are the most valuable DOS services affected by the reentrancy problem. To use a DOS service within an ISR it is necessary to test for a reentrancy conflict prior to executing the service by checking the value of the *in-DOS* flag [21, 55]. This flag is an undocumented feature of DOS and is therefore risky to use because the feature could be dropped on future versions of DOS and there may be undocumented side effects from using this flag. If the in-DOS flag is set, it indicates that the ISR interrupted a DOS service. The ISR will not be able to use any DOS services during this interrupt and must have an alternative method to complete the ISR. The BIOS services are available, but are too primitive to be of value for advanced programming. If possible, it is better to write data to a buffer and after exiting the ISR store or display the information as necessary. The lack of DOS services, use of assembly language, and the requirement that the ISR execute quickly limits the complexity of functions that can be implemented via interrupt-driven I/O.

Implementation method: Before an ISR can be executed it must be installed into memory (Figure 4.4). The ISR can reside in memory in four different forms: as part of the foreground program, as a terminate-and-stay-resident program (TSR) [21, 55], as a device driver [61], or as a BIOS routine [62]. The most common approach is to include the ISR with the foreground program. This is the easiest method and has the advantage that the ISR always follows the application, not needing to be loaded separately during setup or removed after execution. The TSR approach is the simplest way to install the ISR separately from the application program. The device driver is a more complex method of installing the ISR, but has the benefit that device drivers have been standardized by Microsoft as the official method to install software interfaces. The device driver has two disadvantages: access to the ISR occurs in two stages and is thus a little slower to execute, and it is never possible to use DOS services from within the

driver because the device driver is itself a DOS service. The last method to install the ISR is by incorporating it into the ROM-BIOS. The design of the PC anticipated user expansion of the BIOS and dedicated a memory address block for this purpose. To physically install the ROM would require the use of a custom plug-in card. The BIOS approach would find application in a diskless controller and might have the benefit of higher reliability. The TSR, device driver, and BIOS methods have the advantage that once they are installed, they can continue to service the process even when the user needs to access the foreground program to download data or modify the program. In comparison, if the ISR is part of the foreground program or polled I/O is used, the process has to be shut down in order to access the program.

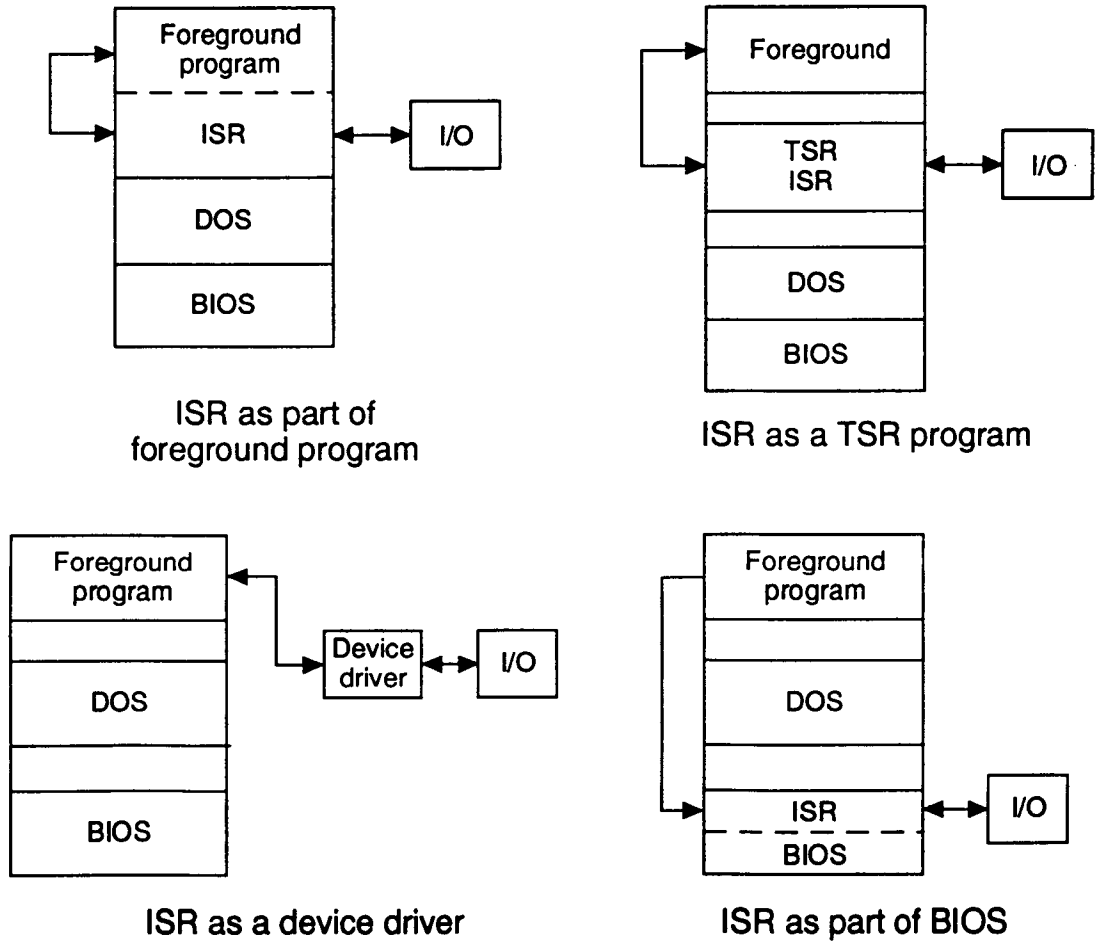

Figure 4.4. Methods of installing ISR into memory.

Data sharing: A disadvantage of interrupt-driven systems is that the sharing and protection of data is difficult. Data must usually be made public to the entire system and this opens the door to accidental corruption.

4.3.3 Multitasking Operating Systems

For applications that only involve a few tasks, interrupt-driven and polled I/O are the best methods to implement the control function. As the number of tasks increase these simplistic methods become too restrictive, limiting the functionality of the control program and making program maintenance difficult. The solution for more sophisticated control applications is to replace DOS with a real-time multitasking operating system (Figure 4.5). A multitasking operating system relieves the programmer from having to be concerned about the timing of task execution and instead allows the programmer to concentrate on the functionality of the individual tasks.

The key element of a multitasking operating system is the task dispatcher which is responsible for scheduling the execution of tasks. There are many different algorithms used to implement multitasking, but they typically use a routine similar to the following: as tasks are created in response to external interrupts or in response to internal requests, they are placed on the execution queue which determines the order in which tasks will run. The task dispatcher is responsible for scheduling tasks according to their priority ranking. When a new task is created, the task manager places the new task on the queue in front of all tasks which have a lower priority. Meanwhile the CPU executes all tasks with equal priority in a round-robin manner via time multiplexing, thus preventing a long task from preventing quick response to other equally important tasks.

In order to insure a predictable response time, a multitasking operating system must be able to perform *preemptive scheduling*. If the task currently being executed is

of lower priority than the task which has just arrived, the operating system should terminate execution of the current task and replace it with the higher priority task. It is the use of preemptive scheduling that insures predictable and bounded response to external events. Not all multitasking operating systems perform preemptive scheduling. If preemptive scheduling is not employed the response time will suffer because a lower priority task may retain control of the CPU for an extended period of time while a higher priority task waits.

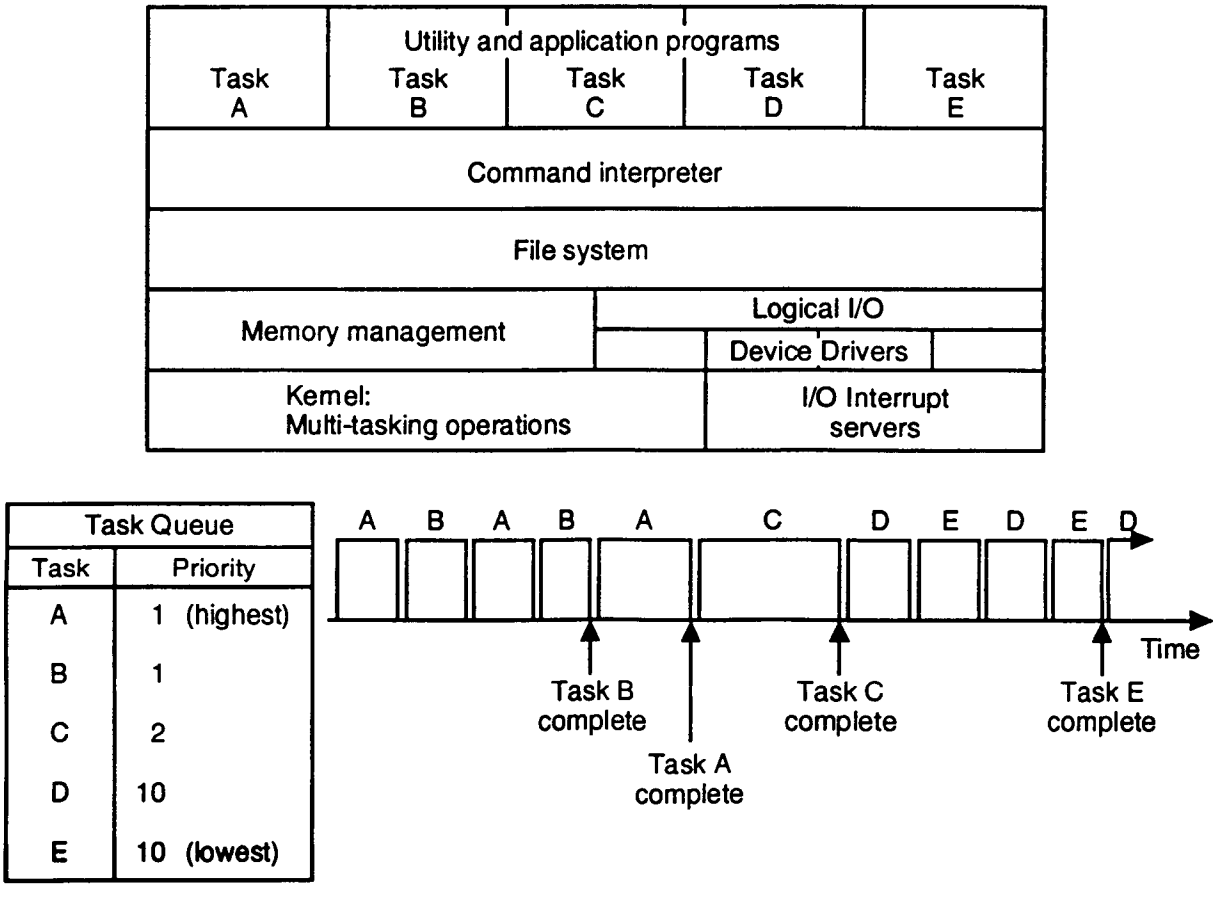

Figure 4.5. Multitasking operating system.

In a real-time operating systems interrupts are processed in two stages. The immediate servicing of an interrupt is handled with an ISR in the same manor that interrupt-driven I/O services interrupts. Once the interrupt request has entered the computer, the task dispatcher evaluates the priority level of the request. If the request is of low priority, it is simply placed on the queue and execution continues with the task that was in progress prior to the interrupt. If the request is of high priority, the task manager suspends the current task and begins executing the task requested via the interrupt.

For the highest performance, care is taken to insure that interrupts are not disabled for extended periods of time. In operating systems like DOS and UNIX, interrupts can be disabled for tens of milliseconds while extended system calls like string operations or block moves are executed. If this is allowed to occur in a real-time operating system the requirement for fast, predictable response time is destroyed. To avoid long periods of latency, these extended services are broken down into stages. The extended service checks back with the operating system after executing each stage to see if a higher priority task must preempt the current execution. In this way the operating system can maintain fast response to high priority tasks.

The various tasks executing in the PC require a method to communicate among themselves in order to share data and synchronize the execution of inter-related tasks. Data sharing between tasks is implemented by allocating a common block of memory for the data. Access to this block is controlled by the memory protection hardware to allow the specified tasks to read and/or write to the block, while prohibiting access by unauthorized tasks. Since time multiplexing of task execution is occurring, a task may start to modify the memory block but not finish the modification during one time slice. If a subsequent task attempted to read this memory, the task would receive invalid data. To prevent this, the operating system uses semaphores as flags to indicate the condition of the data. If the semaphore is zero, the memory is being accessed by another task and the requesting task is placed on a queue. Message passing, another form of the

semaphore construct, is used to provide a method of synchronization between tasks. Messages are placed in a queue called a mailbox while they wait to be received. If tasks are waiting for messages they are placed into a separate queue. The operating system matches up the tasks with the messages, providing a method for one task to signal another in order to synchronize their execution.

Real-Time process control requires the programmer to be able to implement functions that are not included in standard high-level languages. The unique features that are required of a real-time programming language are:

- timely response to real world events
- direct manipulation of hardware resources
- running synchronous processes that must communicate with each other
- having extraordinary error processing and recovery mechanisms for high reliability [59].

To address these needs several real-time multiprocessing languages have evolved. Modula2 is an extension of Pascal and is one of the older multitasking languages. It introduced the concept of the module which encapsulated the data and the procedure to provide security and a higher level of abstraction. Ada, developed for the Department of Defense, is based on Modula2 and was intended for embedded real-time systems. It is a large and complex language with many desirable features and benefits from being defined by a formal specification. The major criticism of Ada is that only experienced programmers can use it safely due to its complexity. The features of C that have made it popular for conventional programming have also made it popular for multitasking operations. But unlike the multitasking languages discussed so far, C does not contain any instructions to handle the unique needs of real-time multitasking. Instead, it is necessary to provide instructions directly to the operating system for these functions. This is a disadvantage because it creates more opportunity for error.

Application: Using a personal computer in combination with a real-time operating system offers the highest performance for control applications. This approach should be considered for applications in which the number of tasks, inter-task communication, and task synchronization requirements are beyond the abilities of the polled and interrupt-driven methods. The advantages offered by a real-time operating system can quickly outweigh the simplicity of polled and interrupt-driven I/O methods for applications of only moderate complexity. (Refer to the difficulties encountered by [59]). If the application appears to border between the use of a real-time operating system and the simpler methods, it would be wise to select the real-time operating system to anticipate unforeseen complexity and to provide the ability to allow for expansion of the control functions.

Complexity: Implementing a control function using a real-time operating system is certain to be more complex than using interrupt-driven I/O. There is always a learning curve associated with a new operating system, and a multitasking system is more complex then conventional systems. Depending on the product selected, the operator interface may be elaborate using a menu system as in the case of OS/2, or very rudimentary like some of the less developed operating systems[63, 64]. The added complexity of using a real-time system is offset by the improved performance that will result.

Response time: Most vendor data sheets rate the response times in the low microseconds. Comparing different operating systems is very difficult as there are no standard methods used. For the highest performance, the best choice is an operating system that does not provide DOS compatibility. If DOS compatibility is important, expect the response time to suffer greatly. This occurs because of the basic limitations present in DOS service routines. Real-Time operating systems compatible with DOS can create the best performance by rewriting the DOS system calls. This approach then leads to a new problem, a lack of true DOS compatibility.

Maximum number of controllable tasks: The use of a real-time operating system should be considered as the number of tasks that require concurrent processing exceeds five. For applications with ten or more tasks, a real-time system is probably the only method to consider. The maximum number of tasks controllable on the PC is limited by the capability of the operating system, typically in the range of 64 to 128 tasks.

Overhead: The use of a real-time operating system adds significant overhead to the process control as compared to implementations under DOS. It has only been with the introduction of more powerful processors like the Pentium that the use of the PC for higher performance control has been feasible. The overhead will be inversely proportional to the services provided. An operating system like OS/2 that supplies good hardware support is also one of the larger and slower systems.

Evaluating operating system choices: The selection of an operating system for a particular application is very difficult, yet it is one of the most critical decisions in the design process[65]. In addition to evaluating the response time and system resources such as disk services, consider the development tools that are available. With software costs being the number one expense in control applications, the lack of good programmer's aids can significantly effect project cost. Another major difficulty in system development concerns the lack of software support of hardware modules. Pay particular attention to the quality of the device drivers supplied with hardware.

CHAPTER 5
EXPERIMENTAL PROCEDURE

5.1 Goals

In order to select the computer system for a particular control application requires a comparison between the throughput of the computer and the control needs of the process. The work presented here is an analysis of the time required to execute programs and the time to perform various storage and I/O functions on the PC. With this analysis, it will be possible to estimate the attainable performance of a process control application running on a PC.

This analysis has focused on three aspects of the personal computer: the performance of peripheral devices controlled by the DOS operating system, the amount of time that is consumed in overhead functions, and the execution rate of C and assembly language programs.

The results consist mainly of measured execution times for various tasks performed by the PC. These measurements are directly related to the hardware platform on which the tests were run. Testing has been performed on 386, 486, and Pentium platforms. (Detailed information on the hardware configurations is contained in the appendix.) To apply these results to a particular application will require a scaling of these performance measures to the new platform.

5.2 Controlling Peripherals

This section focuses on the performance of three common PC peripherals: the hard drive, video monitor, and the parallel port. The access time and the maximum time period that interrupts are disabled has been measured as they relate to applications in real-time control. The results that are presented here are a function of the performance of the peripheral hardware, the performance of the DOS operating system, the speed of the microprocessor, and in some cases, the efficiency of the C compiler that generated the test code. The test programs were all compiled as .COM files; both the code and the data were contained within a single 64 Kbyte segment.

The test setup shown in figure 5.1 was designed to measure the access time of the peripheral under test. The PC was programmed in a loop to continuously exercise a peripheral such as the hard drive. As a means to provide test points to external timing equipment, pin 5 of the parallel port was toggled during each pass of the loop. Pin 5 (peripheral_busy) indicated when the peripheral started an I/O function by going high and indicated the completion of the I/O by going low again. When the peripheral_busy signal was high, the I/O function was in process. The time in which the peripheral_busy signal was high was equivalent to the time necessary for the peripheral to perform an I/O function. The looping program then provided a repetitive signal to display on an oscilloscope. For example, when testing the time to write to the hard drive, the following sequence of events occurs:

write peripheral_busy to high
start writing to hard drive
end writing to hard drive
write peripheral_busy to low
repeat above.

To test for disabled interrupts the test setup of figure 5.2 was used. The PC was operated similar to the setup of figure 5.1 but in addition, an ISR was resident in the test program. A rising edge on hardware interrupt 7 (IRQ7) caused the CPU to vector to the ISR. The ISR executed a small program that output a pulse on pin 3 of the parallel port. Using this test setup it was possible to investigate when hardware interrupts were disabled. With the test program exercising the peripheral in a continuous loop, the oscilloscope displayed the time window in which the peripheral was performing an I/O function on channel 1. Starting with zero delay, channel 2 displayed the isr_on signal, indicating that the ISR was executing and hardware interrupts were enabled. As the delay was increased, the ISR executed at a later time within the peripheral_busy window. As long as hardware interrupts were enabled, the isr_on signal was present. As the ISR was scanned across the window, there were periods of time when the isr_on signal would disappear, indicating that IRQ7 was not being recognized and therefore hardware interrupts were disabled. The maximum length of time that interrupts were disabled, or the "dead time" was the most important quantity being measured. Of secondary importance was the percentage of time that hardware interrupts were disabled.

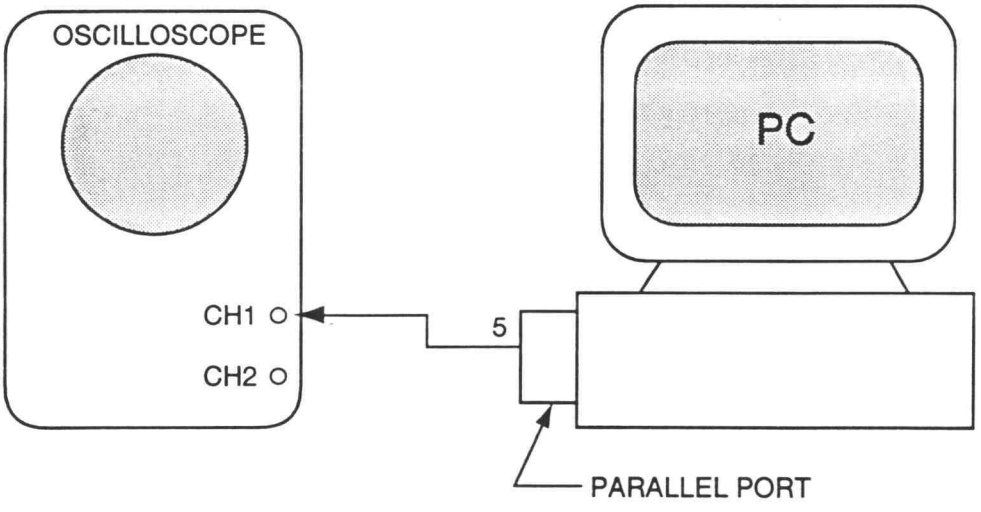

Figure 5.1. Test setup used to measure access time of peripherals.

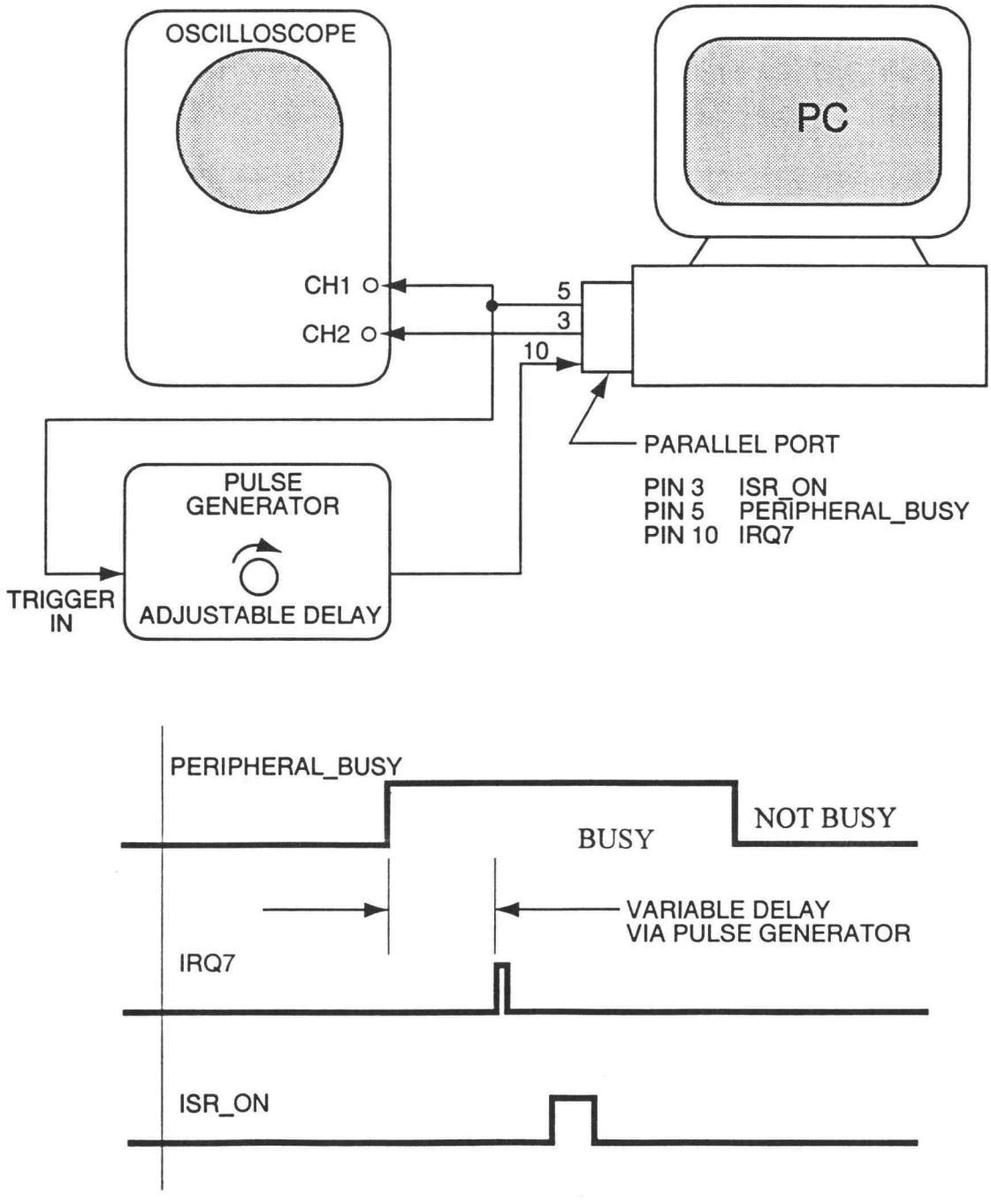

Figure 5.2. Test setup for measuring disabled hardware interrupts.

5.2.1 Hard Drive Performance

The test setups of figures 5.1 and 5.2 were assembled in order to measure the performance of the hard drive. Reading from the hard drive was tested by executing the test program FILEREAD.C. (Refer to program listings in the appendix.) This program continually read files from the drive while also generating the test signal peripheral_busy. The hard drive performance was tested with file sizes of 1K, 100K, and 895K bytes. The results of those tests are displayed in Figure 5.3.

In order for this program to properly test the hard drive performance it was necessary to insure that the file being read was actually residing on the hard drive, and not resident in drive cache memory. If a single file were read repeatedly, the hard drive would pull the file into the cache and then access it directly from the cache. In order to avoid this, DISKREAD.C read ten different files sequentially. For comparison purposes, the time to read a 1 Kbyte file resident in cache memory was 9.6 mS (386 platform). Data is located in a somewhat random order on hard drives due to track and sector locations. To handle this variable, test results show minimum, maximum, and average access times. The time period hardware interrupts are disabled was also measured and is displayed in figure 5.3.

The time required to write a file was also measured using the test program FILE_WRT.C. In this program files were opened, written to, and then closed again. Files from 1 byte to 1 Mbyte in length were measured. The results are given in figure 5.4. The write time for different files did not show as much variation as the read time, but the write time was much longer than the read time. The test for disabled interrupts found interrupts disabled for much longer periods and more often than when reading files as summarized in figure 5.4.

These results give a quantitative measure of the amount of time that will be required to allow reading or writing to the hard drive. For example, as can be seen from

File Size (bytes)	Time Required to Read Hard Drive					
	386 (20 MHz)			Pentium (75 MHz)		
	Min.	Max.	Avg.	Min.	Max.	Avg.
1K	10.0 mS	100.0 mS	35.6 mS	3.5 mS[1]	9.0 mS[2]	8.0 mS
10K	1.0 S	2.0 S	1.4 S	10 mS[3]	32 mS	25 mS
985K	9.8 S	11.2 S	10.4 S	2.0 S	2.2 S	2.1 S

Interrupt Performance (10 Kbyte file)	386 (20 MHz)	Pentium (75 MHz)
Longest Period Interrupts Disabled	15 uS	15 uS
Percent Time Interrupts Disabled	~5 %	~3 %

Notes:
1 Drive cache memory in use
2 First file, therefore no use of cache memory
3 Estimate, unable to observe fastest file access due to test equipment limitation

Figure 5.3. Test results for reading hard drive.

File Size (bytes)	Time Required to Write Hard Drive					
	386 (20 MHz)			Pentium (75 MHz)		
	Min.	Max.	Avg.	Min.	Max.	Avg.
1	300 mS	350 mS	325 mS	680 mS	780 mS	720 mS
1K	-- mS[1]	-- mS[1]	-- mS[1]	620 mS	420 mS	550 mS[2]
10K	300 mS	600 mS	450 mS	380 mS	420 mS	400 mS[2]
985K	3750 mS	4250 mS	4000 mS	2300 mS	2900 mS	2700 mS

Interrupt Performance (1 byte file)	386 (20 MHz)	Pentium (75 MHz)
Longest Period Interrupts Disabled	1.7 mS	<2 mS
Percent Time Interrupts Disabled	~60 %	~5 %

Notes:
1 Unable to measure due to failure of 386 motherboard.
2 Unexplained decrease in access time as files become larger.

Figure 5.4. Test results for writing to hard drive.

the results, it takes about two seconds to read a 1 Mbyte file from the hard drive on the Pentium system, but interrupts are only disabled for 15 uS periods. This time period needs to be contrasted with the requirements of the process control to determine how the PC will perform all tasks in a timely manner.

5.2.2 Video Monitor

The performance of the VGA video monitor was also measured. Using the same test setups as in figures 5.1 and 5.2, the test program SCREEN.C was executed. This program wrote 80 characters to the screen. The results are summarized in figure 5.5. Again the time periods measured here must be contrasted with the requirements of the control system to determine if both tasks can be carried on simultaneously.

Video Performance	386 (20 MHz)	Pentium (75 MHz)
Video Write Time (80 Characters)	21.5 mS	13.4 mS
Longest Period Interrupts Disabled	10 mS	0 mS
Percent Time Interrupts Disabled	~20 %	~0 %

Figure 5.5. Test results for writing to video monitor.

5.2.3 Parallel Port

The parallel port is mapped as a register within the I/O space. For control applications the parallel port can be used as an inexpensive method to provide parallel I/O. The performance measures given here for the parallel port would also apply to any device that is accessed as a register in the I/O space.

The test programs PORT_WRT.C AND PORTREAD.C were used to test the parallel port. This test program was written in C. As a comparison of the time necessary to perform the same task in assembly language, the test programs PRTRDASM.C and PRTWRAS.C were used to write and read the parallel port using assembly language. The interrupts were not disabled during these I/O functions. The results, given in both elapsed time and processor clock cycles, are summarized in figure 5.6. From the Pentium results, it appears that the I/O bus bandwidth is limiting the performance. This access time, though taking many CPU clock cycles, is very fast and not likely to be the limiting factor in simple control problems.

5.2.4 Serial Port

The serial port is the most common interface used in PCs. Unlike the peripherals above, the I/O rate is determined by the baud rate selected for communication. Once a serial port is configured, transmission occurs simply by writing bytes to a register in the I/O space. The time required to write to this register is equivalent to the access time for the parallel port as just discussed. The baud rate is the number of bits per second transmitted and is the rate that the bits are serially shifted out of the I/O register. Standard serial communication requires a header and tail on each

Language	Parallel Port Read and Write Time			
	386 (20 MHz)		Pentium (75 MHz)	
	Read	Write	Read	Write
C	5 uS (100 clocks)	5.5 uS (110 clocks)	2 uS (150 clocks)	2 uS (150 clocks)
Assembly	1 uS (20 clocks)	1.1 uS (22 clocks)	1.8 uS (135 clocks)	1.8 uS (135 clocks)

Figure 5.6. Test results for reading and writing to the parallel port

byte transmitted creating an overhead of 2 to 4 bits (depending on parity and stop bit protocol) for every byte of data. For example, if an interface used 19.2 Kbaud, 8 data bits, no parity, 1 stop bit, one bit would be transmitted every 52 uS. When the overhead is taken into account, the transfer rate is:

RS-232 19.2 Kbaud serial transfer rate =

19.2 K bits/second * 1 byte data/ (8 + 2) total bits =

1920 bytes/second or 1 byte/521 uS.

In this example, the control program would need to write to the serial port one byte every 521 uS in order to keep the serial line from going idle. The CPU has plenty of time to perform other tasks. Therefore, in this example the baud rate is limiting the I/O rate of the computer system. This is an example of how the serial port is an extremely slow transfer method. The serial port can not be used for demanding, high speed applications.

5.3 Operating Overhead

There are two miscellaneous functions performed routinely by the PC that could have an effect on applications involving real-time control: the servicing of the timer and the amount of time needed to perform a context switch.

The time-of-day clock used in personal computers is implemented with a combination of hardware and software functions. The 8253-5 clock/timer chip provides a countdown lasting 54.9 mS. Each time the timer completes a countdown, it interrupts the CPU using hardware interrupt number 8. The CPU services the timer using an ISR that resets the counter and adds 54.9 mS to the memory location storing the time of day. This is continuously occurring during normal computer operation. Therefore 18.2 times per second (1/54.9 mS) this ISR is executed by the PC. This ISR took 48 uS to execute on the 386 and 21 uS on the Pentium platform (refer to TIMER.C). In applications

involving high speed control it would be necessary to consider how this ISR will affect system timing and if a conflict exists, the timer may need to be disabled.

Whenever a computer is required to execute an ISR, the current state of the CPU must be stored in order for the CPU to be able to return to the current task upon completion of the ISR. This change of state is referred to as a "context switch". The time required to perform a context switch can be of importance if the context switch time approaches the execution time of the ISR.

In the personal computer when a hardware interrupt is received, the sequence of events leading up the context switch involves a series of handshakes between the microprocessor and the programmable interrupt controller (PIC). The following sequence of events occurs:

1) The 8259A PIC screens the interrupt for masking and priority as predefined during initialization of the PIC. If the interrupt line within the PIC has been enabled (unmasked) during initialization, and there are no higher interrupts pending, the interrupt is relayed to the interrupt (INT) line of the microprocessor.
2) If interrupts are enabled within the CPU, the CPU checks the interrupt status at the end of each instruction.
3) If an interrupt is present, the CPU pushes the flags register onto the stack.
4) The CPU disables further interrupts within the CPU.
5) The CPU pushes the code segment (CS) and instruction pointer (IP) onto the stack.
6) The CPU acknowledges the interrupt by raising the interrupt acknowledge (INTA) line of the PIC.
7) The PIC places the interrupt number on the data bus.
8) The CPU reads the interrupt number off the data bus.

9) The CPU calculates the entry number for the interrupt vector table based on the interrupt number received (entry number = interrupt number * 4).
10) The CPU reads the address contained in the interrupt vector table corresponding to the interrupt number.
11) The CPU loads the new address into the CS and IP registers.
12) The CPU jumps to the ISR located at CS:IP.
13) The ISR is now executing and the first task of the ISR is to save the current state of the machine by pushing the registers onto the stack.

The entire sequence above must occur before any servicing within the ISR can occur. The time to perform the above tasks was measured using the test program CONTEXT.C. This test program measures the time to enter the ISR including the time to save the registers of the machine. The test does not measure the time to save the context which is performed within the ISR code. The 386 took 6.5 uS and the Pentium 3.4 uS. (For this test, overhead needed to toggle the parallel port used for timing purposes was significant and was subtracted out of the results. The overhead was measured using the test program CONTXTOH.C.)

5.4 Program Execution Rates on the PC

Another issue that affects the performance of the PC-based control system is the rate at which lines of program code can be executed. A typical control program will collect a measurement from the controlled process and then compute the next output value based on a mathematical or logical algorithm. Therefore, in addition to performing fast I/O the control computer must be able to execute program code quickly. The analysis presented here provides a method to estimate the time required to execute program code written in either C or assembly language.

A well known problem present when measuring the performance of processors is selecting an instruction mix for testing. The actual instruction mix selected can skew the results in either direction based on the complexity of the instructions used. For example, an instruction set biased toward instructions such as moves and logical manipulations will execute much faster than a set containing a majority of floating point operations. The goal of the instruction mix should be to simulate code used in real-world applications. Unfortunately, this goal is not easily met due to the variation of instruction content in real world programs. This variation is perhaps even more pronounced in control applications.

5.4.1 Execution Rate of C Code

The ideal method to calculate the execution rate of C code on the PC would be to obtain the assembly language translation of each C instruction, and then analyze the execution rate of the resulting assembly code. Unfortunately, the manufacturers of C compilers do not provide this information. Therefore, a different approach must be taken. The method chosen here was to count the lines of C code in a sample program and then measure the time required to execute that program. This provides a simple measure of the computer performance while avoiding the problem of access to compiler translation code.

Obtaining a measure of the code execution rate for a high level language in this method is complicated by both the instruction mix problem and the method of counting the lines of code. One example of the line counting problem is evident when a loop instruction is encountered. If the loop instruction is counted as a single line of code, the execution rate is a function of the number of loops, and for an infinite loop, which is common in control application, the execution rate would be zero. The solution for counting lines of code that involve loops is to count each pass through the loop separately. Another variable that enters the calculation of the number of lines of code

occurs when a library function is used. Here, one line of code will cause the compiler to link an external procedure to the program. In the case of linked library functions, the software vendor does not provide the source code of the library function, and therefore the size of the function can not be determined. These library calls add significant execution time to the program, but appear as a single line of code. In this instance, the most appropriate solution is to count the library function as a single line of code, realizing that this adds an unavoidable variable to the measurements.

Figure 5.7 presents the results of C execution rates for several programs. The description column defines what was present in each program. Program 1 and 2 were essentially the same program with the library functions removed for program 2. Therefore, the increase in execution rate was due to the removal of the library function. Program 3 was a completely different program, and although it contained 2 library functions, ran considerably faster. This difference is an example of the variation resulting from differing instruction mixes. The results are given in clock cycles per line of C code. Using clock cycles/line removes the clock rate parameter from the performance measure. Therefore, the differences in the average execution rate are due to improved microprocessor architecture in the more advanced designs. While these three programs exhibit a wide ranged of execution rates, the results are still useful for estimating the time needed to perform an algorithm computation in a control

File Name	File Description	Execution Rate (Clock cycles per line of C code)		
		386 (20 MHz)	486SX (33 MHz)	Pentium (75 MHz)
C_RATE_1	File transfer program with library calls	250	166	87
C_RATE_2	File transfer program with no library calls	208	58	38
C_RATE_3	Drystone benchmark	73	17	9.5
	Average Execution rate	177	80	45
	Normalized Execution rate	100%	45%	25%

Figure 5.7. Execution rate of C code for various programs.

application. It should be noted that these programs did not perform any I/O functions. I/O was purposely omitted from these test programs due to the work presented early giving I/O times.

5.4.2 Execution Rate of Assembly Language

Calculating the execution rate of assembly language provides the opportunity to perform more accurate studies than are possible with high level languages since the uncertainty associated with the high level compiler is eliminated. For this analysis an instruction mix was selected and then the theoretical execution rate was calculated based on clock cycle usage information provided by Intel. To check the results, test programs were written that implemented the instruction mix.

When measuring the execution rate of assembly language programs, the selection of the instruction mix is still an issue just as in the C timing presented earlier. For this testing the instruction mix selected was based on work by Adams and Zimmerman[66]. In their work, Adams and Zimmerman monitored the execution of Turbo C, MASM, and Lotus programs running on the PC and tabulated the instruction usage. This is probably a good approach to defining an instruction mix for an office environment, but one would wonder how this compares to an instruction mix used in control applications. Nevertheless, Adam and Zimmerman's instruction mix was used for this analysis.

Figure 5.8 lists the most commonly used instructions and gives the average instruction usage of all three programs as calculated by [66]. The clock cycle usage for each instruction per Intel's *Programmer's Reference Manual*[67] is given in the third column. The clock cycle usage includes two numbers for conditional instructions: one for the branch taken and the other for the branch not taken. The *m* in several entries is associated with the conditions under which the current instruction is being executed.

Notes:

1. Average clock cycle usage is based on work by [66]. List includes instructions used more often than 1.5%.
2. Average clock cycle usage = (average usage) X (instruction clock cycle usage).
3. "m" is dependent on conditions in which instructions are executed.
   Consult [68] for details.
4. "r/m" refers to ratio of register usage to memory usage.
5. The theoretical performance is for the 386 processor and assumes the following test conditions:
   - instruction has been prefetched and decoded and is ready for execution
   - bus cycles do not require wait states
   - no logical bus hold requests delaying processor access to bus
   - no exceptions are detected during instruction execution
   - memory operands are aligned
   - if an effective address is calculated, it uses use 2 general purpose registers (see [46?], Volume II, p. 5-380)
   - operating in real mode (note: task switching takes a long time)
   - all branch/jump destinations are coded as immediate data

Figure 5.8a. Instruction Mix and Clock Cycle Usage for Intel Processors

Instruction	Avg. Usage[1]	Clock Cycle Usage m = memory, r = register	Average Clock Cycle Usage[2]	Assumptions[3]
JCC (jmp conditional)	10%	7+m (m=2)[4] if taken 3 not taken	(5% X 9) + (5% X 3) = .60	1/2 conditional jumps taken, 1/2 conditional jumps not taken
CALL, CALLF	4%	7+m (m=2)	(4% X 9) = .36	1/2 taken, 1/2 not taken
RET, RETF	4%	10+m (m=1)	(4% X 11) = .44	
LOOP	4%	11 + m (m = 2)	(4% X 13) = .52	
JMP	2%	7 + m (m = 2)	(2% X 9) = .18	
CMP	7%	r/m[5]= 2/5	(4% X 2) + (3% X 5) = .23	4/7 register compare 3/7 memory compare
SAL, SHR, RCR	5%	SAL & SHR r/m = 3/7 RCR r/m = 9/10	SAL/SHR (2% X 3) + (1% X 7) = .13 (1% X 9) + (1% X 10) =.19	instruction mix ratio: 3/5 SAL & SHR, 2/5 RCR register/memory usage SAL & SHR 2/3 r, 1/3 m RCR 1/2 r, 1/2 m
ADD	3%	r/m = 2/7	(2% X 2) + (1% X 7) = .11	register/memory usage 2/3 r, 1/3 m
OR, XOR	3%	6	(3% X 6) = .18	r to m ORing
INC, DEC	3%	2	(3% X 2) = .06	register usage only
SUB	2%	r/m = 2/7	1% X 2) + (1% X 7) = .09	1/2 immediate to r 1/2 immediate to m
CBW	1%	3	(1% x 3) = .03	
TEST	1%	2	(1% X 2) = .02	immediate to r
MOV	27%	r/m = 2/4	(13 X 2) + (14 X 4) = .82	13/27 r to r, 14/27 m to r
LES	3%	7	(3% x 7) = .21	
PUSH	7%	r/m = 2/5	(4% X 2) + (3% X 5) = .23	4/7 r, 3/7 m
POP	5%	5	(5% X 5) = .25	100% m
TOTALS	91% of instruction usage		4.65 clocks per instruction (considering 91 % of instruction usage) 4.65 X 100/91 = 5.11 clocks per instruction (average)	

Figure 5.8b. Instruction Mix and Clock Cycle Usage for Intel Processors

Consult the Intel data book [68] for details. The clock cycle usage quoted by Intel is valid under conditions that in essence create a "best case" condition. The notes given at the bottom of figure 5.8 specify the conditions under which the Intel values are valid.

The forth column gives the average clock cycle usage and is derived by multiplying the average usage times the clock cycle usage. This is the theoretical percentage of time the PC would be executing each particular instruction.

Several assumptions concerning the ratio of *taken* to *not taken* branches, register versus memory usage, and the ratio of each instruction within a particular instruction category were necessary for this testing. These assumptions are defined in column 5 of figure 5.8.

The instructions listed account for 91% of the execution time in this instruction mix, with the remaining time using instructions with less than 1.5% occurrence. The average clock cycle usage of the 91% is obtained by summing column four of figure 5.8. The result, 4.65 clock cycles per instruction, is adjusted up to 100% by multiplying by 100/91.

$$4.65 \text{ X } (91/100) = 5.11 \text{ clock cycles per instruction.}$$

This performance measure is a function of the efficiency of the processor architecture and does not include the clock rate. Therefore these results are easily scaled to any processor clock rate. The clock cycles per instruction information is from the Intel 386 data book and only applies to that processor architecture. Testing has been performed that will compare the 386 architecture to the 486 and Pentium processors.

To compare the calculated performance to the measured performance, the test program INST_MIX.ASM was used. For the instructions in table 5.8, INST_MIX.ASM executes each instruction repeatedly up to the value corresponding to average instruction

usage. The program executes 91 instructions total, and the execution time was measured in order to calculate the execution rate. For more accuracy, the overhead used by the test program to signal the start and stop of testing was subtracted out. The results are shown in figure 5.9. From the results it shows that the measured performance was almost a factor of two poorer than the theoretical performance.

In order to take a closer look at why the measured performance did not compare to the calculated performance, a more detailed test method was used. This revised test presented in figure 5.10 measured the performance of each instruction separately. For each instruction, a test program was written that executed that instruction 100 times in a row. This approach was used for two reasons. The first concern was that the instruction pipeline was not being used efficiently. Intel's conditions for calculating the performance assume an uninterrupted pipeline. The second reason for testing each instruction separately was to allow an instruction-by-instruction analysis of the measured vs. theoretical performance.

Looking more closely at figure 5.10, the instruction set and the average instruction usage values from figure 5.8 are repeated in columns one and two. The execution time was measured, the equivalent clock cycle usage calculated, and the

Architecture	Performance (disturbed pipeline) (average clk. cycles per assy. instruction)
386 theoretical performance	5.1 (51%)
386 measured performance	9.9 (100%)
486SX measured performance	4.2 (42%)
Pentium measured performance	3.2 (32%)

Figure 5.9 Assembly language code execution rate.

Instruction	Average Usage[1]	386DX Measured Clock Cycle Usage	386DX Avg. Clock Cycle	486SX Measured Clock Cycle Usage	486SX Avg. Clock Cycles	Pentium Measured Clock Cycle Usage	Pentium Avg. Clock Cycles	Test Program
JCC (jmp conditional)	10%	7.2	.72	12.5	1.3	7.5	.75	COND_JMP.ASM
CALL, CALLF	4%	9.3	.37	4.6	.18	1.6	.06	CALL&RET.ASM
RET, RETF	4%	9.3	.37	4.6	.18	1.6	.06	note 2
LOOP	4%	12.5	.50	6.9	.23	10.5	.42	LOOP.ASM
JMP	2%	10.8	.22	5.0	.10	1.9	.04	JMP.ASM
CMP	7%	6.45	.45	2.0	.14	2.0	.14	CMP_MIXD.ASM
SAL, SHR, RCR	5%	10.0	.50	5.6	.28	4.0	.20	SHIFT.ASM
ADD	3%	3.7	.11	5.0	.15	4.8	.14	ADD_MIXD.ASM
OR, XOR	3%	21	.63	2.0	.06	7.5	.22	OR.ASM
INC, DEC	3%	2.0	.06	1.1	.03	4.8	.14	INC.ASM
SUB	2%	11.6	.23	5.4	.11	5.0	.10	SUB.ASM
CBW	1%	2.9	.03	3.2	.03	3.1	.03	CBW.ASM
TEST	1%	2.2	.02	1.2	.01	0.6	.01	TEST.ASM
MOV	27%	2.2	.60	1.1	.30	1.1	.30	MOV.ASM
LES	3%	8.5	.26	5.9	.18	4.2	.13	LES.ASM
PUSH	7%	2.5	.18	0.8	.06	0.4	.03	PUSH&POP.ASM
POP	5%	2.5	.12	0.8	.04	0.4	.02	note 3
TOTALS	91% of instruction usage		5.37 X (100/91) = 5.9 clock cycles per instruction (ave.) (100%)		3.38X (100/91) = 3.7 clock cycles per instruction (ave.) (63%)		2.79X (100/91) = 3.1 clock cycles per instruction (ave.) (52%)	

Notes:

1. Instruction mix is per [86] as in figure 5.8. 2. Measurements for call and return were combined. 3. Measurements for push and pop were combined.

Figure 5.10. Measured execution rate of individual instructions on the 386.

resulting measured clock cycle usage was obtained as presented in column three. To calculate the average clock cycle usage for this instruction mix, the average usage was multiplied by the measured clock cycle usage and entered in column four. As in figure 5.8, the average clock cycle usage is summed, and adjusted up to 100% and a new measured average clock cycle usage of 5.9 clocks per instruction was obtained.

As can be seen from the test results in figure 5.10, the measured execution rate of 5.9 clock cycles per instruction compared well with the theoretical value of 5.11 cycles per instruction. The improved correlation is believed to be a result of using the pipeline more efficiently. Also of interest is the higher performance of the 486 and Pentium architectures.

The normalized performance is summarized in figure 5.11. There are two interesting points to notice. The 386 performance improves significantly going from the disturbed to the undisturbed pipeline while the Pentium shows little change. This indicates the Pentium pipeline design is more robust and tolerant of disturbances. Also note that the C code performance tracks the disturbed assembly code performance, indicating typical C code also disturbs the pipeline.

The calculations presented in figure 5.8 provide a method to calculate the execution rate of assembly language programs for control applications. The testing presented in figures 5.9 and 5.10 verify this performance and bring up a practical aspect of the Intel performance. When the pipeline was not fully utilized, as in the testing of figure 5.9, the performance decreased by a factor of 2 for the 386. The lower performance is probably more typical of real world performance and certainly needs to be anticipated in a control application.

Architecture	Performance (execution time normalized to 386)		
	Disturbed Pipeline	Undisturbed Pipeline	C Code
386 theoretical performance	100%	100%	----
386 measured performance	194%	116%	100%
486SX measured performance	82%	72%	45%
Pentium measured performance	63%	61%	25%

Figure 5.11. Normalized performance of Intel architectures.

5.5 Estimating the Performance of a Control System

With the work that has been presented in this chapter, it is now possible to estimate the performance of a PC-based real-time control application. The performance measure will be dependent on the application and since each application uses a different performance measure, the method of analyzing the performance will also vary.

Guidelines are presented here that will provide a general approach for estimating the performance of a PC-based control system. Because the method to estimate the expected performance will depend on the specific control application, it would be difficult to provide a single formula for all applications. Instead guidelines will be given and an example will demonstrate the method.

1. Identify the basic computational requirement of the system: The first quantity to identify is a basic requirement of the control system that must be performed. There will be some aspect of the control function that must be performed that is fixed and is not available for modification. For example, in a communications application, there might be a required manipulation of the communication packet. In a DSP

application, there might be a required frequency response of the digital filter. In a multi-point control application, the complexity of the control algorithm may be fixed.

2. Identify the performance measure that is a function of the control computer: The next aspect to identify is the performance measure that will be dependent on the control computer. This is the quantity that will vary depending on how fast the computer can perform the basic requirements identified in guideline 1. In the communications example, the performance would be measured by the data transfer rate. In the DSP example, the performance would be based on the complexity of the filter that can be implemented. In the multi-point control example, the performance would be based on the number of controllable points.

3. Scale these performance measures to the new platform: Since the computer used for the control application will differ from the one used in this research, the execution speed of the new computer must be estimated. For each function needed in the control application, scale the execution times to the new platform. With the results of the 386 and Pentium testing, it is possible to interpolate or extrapolate the data as necessary to apply to other platforms. If more accurate correlation is needed, the actual testing performed here can be repeated on the platform in question.

4. Estimate the maximum performance attainable: Using the scaled performance measures, calculate the time required to perform the computations identified under guideline 1. Using the performance measure identified in guideline 2, calculate the estimated performance. This is typically the inverse of the execution time. In the communication example, after calculating the execution time to input the packet, perform the required packet manipulation, and output the packet, the data transfer rate can be obtained as the inverse of the execution time. In the DSP example, after calculating the time required to update the filter coefficients, the maximum sample rate is the inverse of the update rate. For the feedback loop example, after estimating the time required to calculate the new control values and update the feedback signal, the

performance would be the inverse of the execution time. It is suggested that a computing safety margin of 5 to 10 be maintained at the early design stage to compensate for the inaccuracies of the performance analysis and to allow room for incorporating unexpected tasks or design improvements.

For synchronous applications, once the computational time is known, a relatively exact estimate of the performance can be obtained. On the other hand, many applications involve an asynchronous response to a system. In this case an additional performance variable arises. In asynchronous systems, the analysis must go beyond the execution time and consider the probability of completing the computation, the average idle time waiting for request for service, the consequence of missing a response, etc. This is beyond the scope of this work but is important to consider in real-world applications.

Any particular control application can be approached in a variety of ways. For example, in the DSP application the engineer could select the filter order (complexity) to be the basic computational requirement and allow the frequency response to be the performance measure. The point here is that there are many approaches to a control problem. The intent of these guidelines is to present a concept, and not attempt to define an exact method.

5.6 Design Example

As a demonstration of the method proposed for estimating the performance of the PC in a control application, a communication task between two PCs will be analyzed. The data transfer rate will be estimated and the result will be compared to the measured performance.

In this example, it is desirable to provide communication between two PCs via their parallel ports. This communication link will have programs running simultaneously on two machines, with one machine configured as a transmitter and the other as a receiver. The transmitting machine will use a foreground task (figure 5.12) to generate data and store the data in a circular buffer, calling an ISR (figure 5.13) via a software-generated interrupt. The ISR handles low-level transmission, pulling data off of the circular buffer, sending it to the parallel port, and triggering the receiver IRQ7. The receiver foreground task (figure 5.14) will read the data from the circular buffer and display it on the screen after the ISR residing on the receiver (figure 5.15) captures the data and places it on a circular buffer.

1. Identify the basic computational requirement of the system: The first step in the analysis is to identify the basic physical limitations of the application that are not available for modification. For this example, it will be assumed that using the parallel port, generating the data on the transmission side, and displaying the data on the receiver side is required.

2. Identify the performance measure that is a function of the control computer: In this application the performance measure will be the rate at which data can be transferred between machines.

3. Scale the performance measures to the new platform: In this application the transmitter will use a 50 MHz 486DX based PC. It is therefore necessary to scale the measured performance parameters to this new platform. Figure 5.16 provides a summary of the performance measures of interest and an estimation of the 486DX performance based on performance of 386, 486SX, and Pentium platforms.

The receiver will use a 75 MHz Pentium based PC. Therefore it will not be necessary to scale performance for this platform.

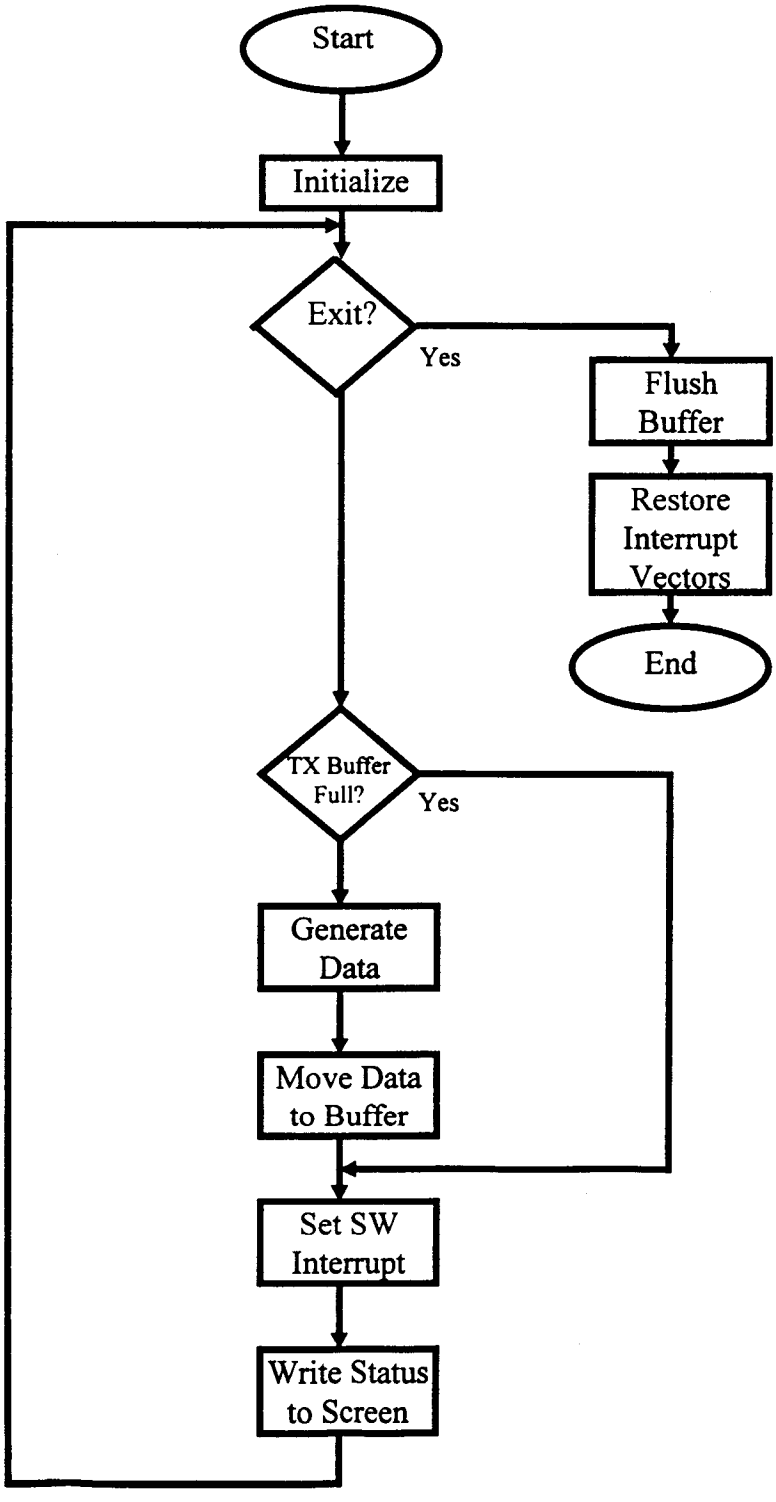

Figure 5.12. Flowchart-transmitter foreground program.

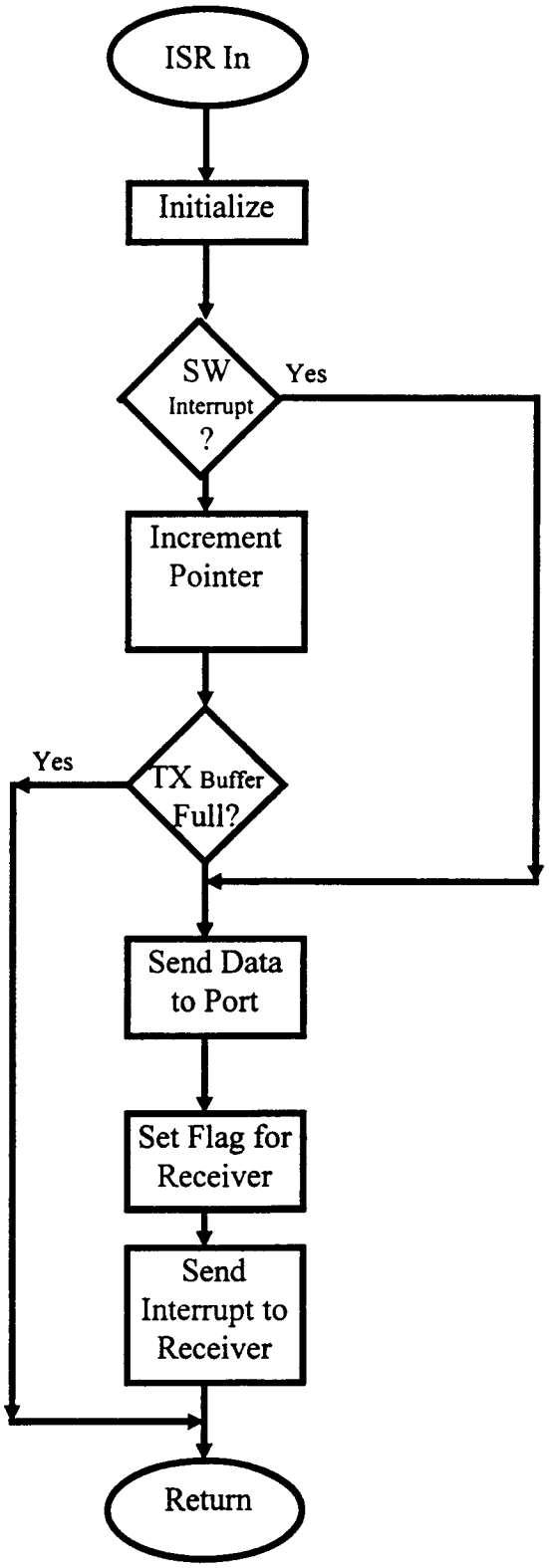

Figure 5.13. Flowchart-transmitter ISR.

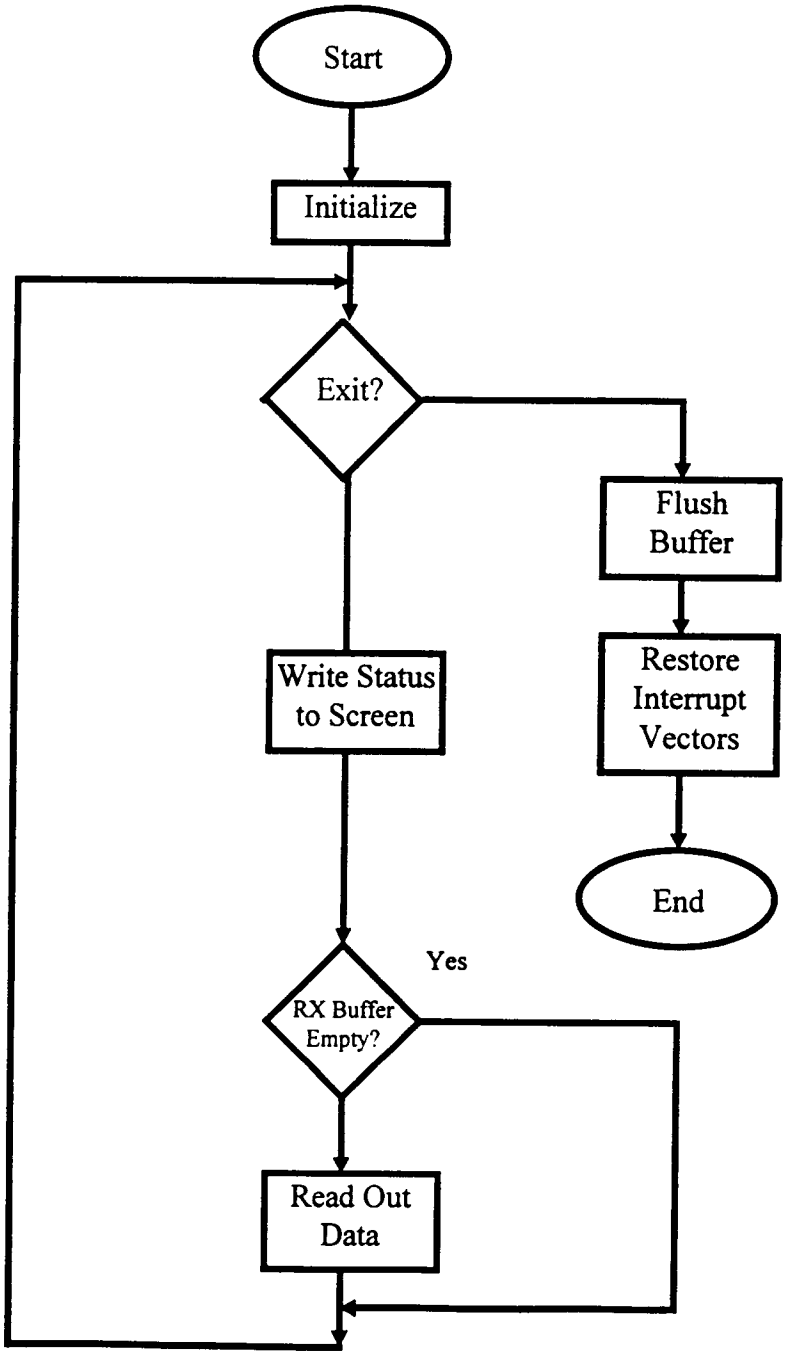

Figure 5.14. Flowchart-receiver foreground program.

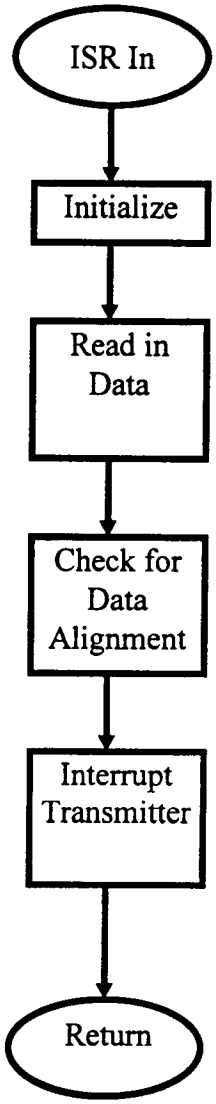

Figure 5.15. Flowchart-receiver ISR.

Performance Measure	386 (20 MHz)	486 SX (33 MHz)	**486 DX (50 MHz) Estimated**	Pentium (75 MHz)
C code execution rate (lines/clk)	177	80	80	45
Assy. code execution rate (inst./clk)	5.9	3.7	3.7	3.1
Parallel port access time (uS)	1.1	--	2	1.8
Video write time (80 char.) (mS)	21.5	--	15	13.4
ISR context switch (uS)	6.5	--	5	3.4

Figure 5.16. Estimated performance for 486DX 50 MHz.

4. Estimate the Maximum Performance Attainable: In order to estimate the performance of an application such as this communication program requires an analysis of each of the simultaneous processes occurring in the computers. There are four asynchronous processes running together as depicted graphically in figure 5.17. The transmitter foreground program is generating data and placing bytes on the circular buffer. The transmitter ISR is pulling bytes off of the circular buffer and sending them to the parallel port. The receiver ISR is capturing data from the parallel port and placing them on the circular buffer and the receiver foreground program is pulling data off of the circular buffer and displaying the results.

In order to estimate the performance, the execution time of each of these processes must be estimated as follows:

Transmitter foreground program:

100 lines of C code X 80 clocks/line X 1/50 MHz clock = .16 mS

20 characters to screen X 15 mS/80 char. (interpolated) = 3.75 mS

TOTAL 3.9 mS

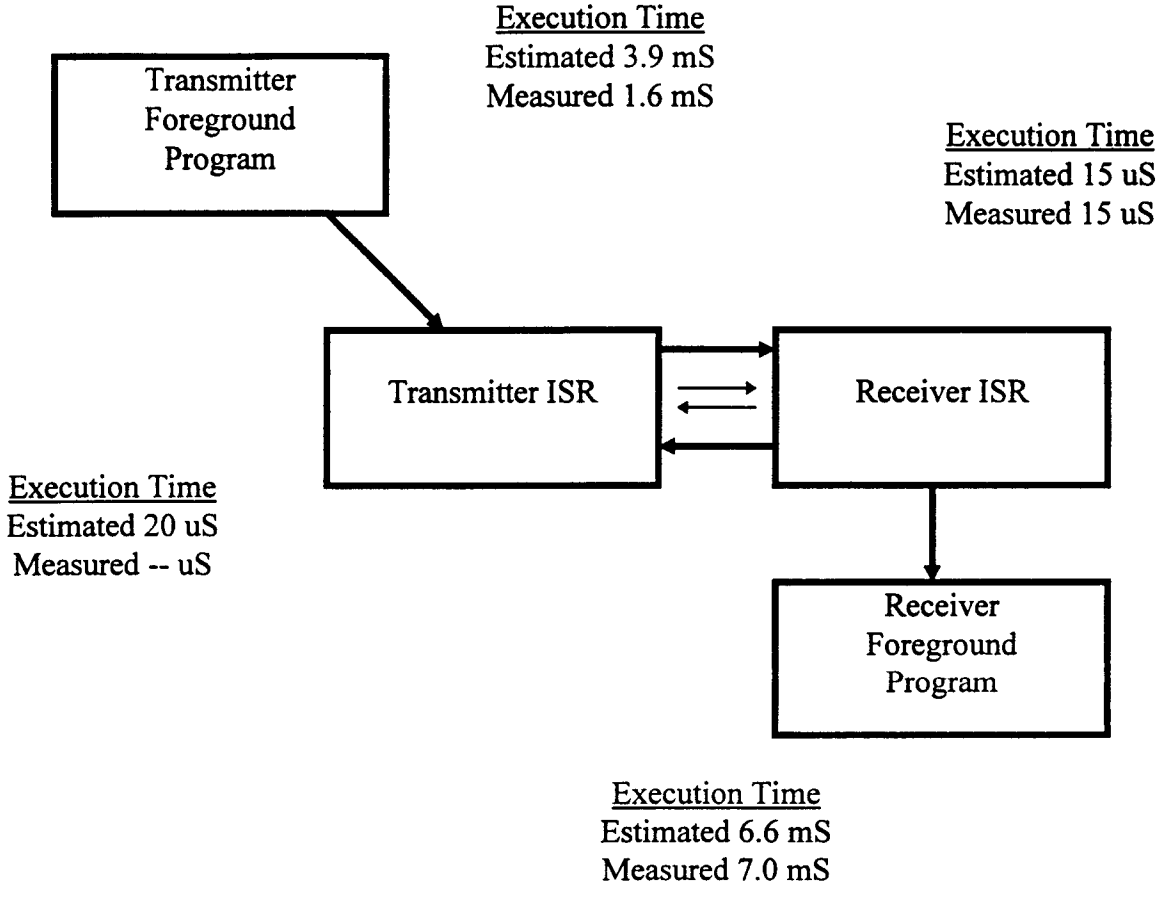

Figure 5.17. Asynchronous processes of communication program.

Transmitter ISR:

context switch	= 5 uS
100 lines assy code X 3.7 clks/instr. X 1/50 MHz clock	= 7.4 uS
4 parallel port accesses X 2 uS/port access	= 8 uS
TOTAL	20 uS

Receiver foreground program:

100 lines of C code X 45 clocks/line X 1/75 MHz clock =	.06 mS
40 characters to screen X 13 mS/80 char. (interpolated) = 6.5 mS	
TOTAL	6.6 mS

Receiver ISR:

context switch	= 3.4 uS
100 lines assy code X 3.1 clks/instr. X 1/75 MHz clock	= 4.1 uS
4 parallel port accesses X 1.8 uS/port access	= 7.2 uS
TOTAL	15 uS.

With each of the four execution times estimated, it is now possible to estimate the maximum obtainable performance. From figure 5.17, it is obvious that the execution times of the ISR are insignificant. Therefore the foreground programs will dominate the performance. Based on the receiver execution time of 6.6 mS, the maximum transfer rate is estimated at 150 nibbles/second (1/6.6 mS).

The measured transmission rate was 120 nibbles/second. For a more detailed analysis of the accuracy of the estimates, the performance of each of the processes in figure 5.17 was measured to compare with the estimated performance. The largest error was a factor of 2.4 in the transmitter foreground program. This is within the expected accuracy of this technique and points to the requirement of allowing a factor of 5 to 10 in computer performance headroom during the design phase.

The programs used throughout this section along with detailed documentation are available from the author for any interested parties.

CHAPTER 6
CONCLUSION

6.1 Summary

The work presented here has been an attempt to provide a method to estimate PC performance during the design phase of a process control project. With this analysis, it is possible to determine what functions of a process can be controlled and at what performance levels. Designers will have a good grasp of the timing issues in a PC-based control application prior to entering the lab to start prototype testing. This method will be most applicable when a small number of I/O points needs to be controlled with a relatively high bandwidth as compared to using a real-time operating system.

This method differs from conventional performance analysis in that basic computer functions are measured to form a building block approach. With these building blocks, performance estimates of complex applications can be analyzed by assembling a group of basic computer functions. There is an added benefit in that it provides the designer with more insight into PC performance. With the performance of each basic function of the PC quantified, the designer can modify the control approach as necessary to avoid computing bottlenecks that might otherwise limit overall performance. This method of viewing PC performance will give the designer a new tool for evaluating throughput issues. The designer can compare measured performance to estimated performance and pinpoint which functions are consuming more CPU time than anticipated. This can be particularly helpful during the debugging phase, when

unexpected delays in software modules can be traced to malfunctions and misconceptions about computing rates.

The work presented here highlights the common problem of I/O bottlenecks in computer architectures. This can be seen in the of the performance of the Pentium as compared to the 386. The code execution rate of the 75 MHz Pentium is 15 times faster than the 20 MHz 386, but both computers write to the parallel port at approximately the same rate. The PC industry is addressing I/O performance with faster busses and smarter I/O chips, but for control applications it appears that I/O bandwidths will continue to limit the ultimate performance.

6.2 Limitations

This work will have limitations in more complex control applications. As the size of tasks becomes larger, the estimates necessary for this approach will not have the accuracy to provide meaningful results. This method can still be useful for providing a first cut at estimating performance and then, along with laboratory measurements, the estimates can be corrected to keep the accuracy within reasonable limits.

6.3 Future Work

This thesis work has provided the basic foundation for a new approach to sizing a control application to the PC. From the work presented here, opportunities for further research in this area have become evident. Some of the more interesting areas are summarize below:

- Provide a more comprehensive set of software and hardware characterizations. For example, graphics, DMA transfers, BIOS performance, and a wider selection of PC platforms.
- Create a standard test for PC platforms that could be universally used to allow market-wide evaluation of vendor products. Integrate testing into an application program for automated performance measurement.
- Compare these measurements with other benchmark tests and vendor data sheets to provide correlation of results. Standard Performance Evaluation Corporation (SPEC) [69] and PC magazine provide test suites for quantifying computer performance.
- Investigate the use of the PC for high-reliability redundant computer systems. The low cost of the PC solves a significant hurdle typically found in redundant system design.
- Perform analysis of instruction mixes used in control applications. Contrast this instruction mix with that commonly used in the office environment and investigate the differences in performance that can be expected.
- Use the techniques of chapter 5 to investigate the architectural performance of different processor families.

REFERENCES

[1] S. Tzafestas and J .K. Pal, eds., *Real Time Microcomputer Control of Industrial Processes,* Kluwer Academic Publishers, Dordrecht, The Netherlands, 1990.

[2] S. Bennett, *Real-Time Computer Control: An Introduction,* Prentice Hall International, Hertfordshire, UK, 1988.

[3] S. Savitzky, *Real-Time Microprocessor Systems,* Van Nostrand Reinhold, New York, NY, 1985.

[4] S. Bennett and D. A. Linkens, eds., *Real-Time Computer Control,* Peter Peregrinus Ltd., London, UK, 1984.

[5] D. J. Larin, "Cell Control: Widespread Use Awaits Technology Improvements," *Chilton's I & CS,* Vol. 61, No. 10, Oct. 1988, pp. 63-65.

[6] E. Bassett, "Detroit Auto Makers Driving Real-Time PC Control," *In Tech,* Vol. 42, No. 4, April 1995, pp. 85-89.

[7] D. Shear, "Embedded Systems Developers Embrace the PC Architecture," *EDN (European Edition),* Vol. 39, No. 5, March 3, 1994, pp. 39-40, 42, 44-45.

[8] W. Crawford, "PC Price and Performance: The Values Just Keep Improving," *Library High Tech,* Vol. 13, No. 1-2, 1995, pp. 123-144.

[9] J. Purvis, "The Personal Computer as Instrument Controller," *Test and Measurement World,* Vol. 6, No. 2, Feb. 1986, pp. 42-51.

[10] J. Pinto, "Industrial Microcomputers Gain Still More Ground," *Chilton's I & CS*, Vol. 61, No. 11, Nov. 1988, pp. 27-29.

[11] Keithley Metrabyte, "Microcomputer Busses-Overview," *Tech Tips #2*, pp. 1-14.

[12] Data Translation, "Retrospective," *1990 Data Acquisition Handbook*, pp. 13.25-13.26.

[13] Data Translation, *1996 Data Acquisition Handbook.*

[14] Keithley Metrabyte, *Data Acquisition & Control,* 1996.

[15] Analogic, *Full Line Catalog,* 1996.

[16] Gage Applied Scientific, *Databook* 1995.

[17] Industrial Computer *SourceBook,* 1996.

[18] R. Midyett, "Industrial PC Software: The Driving Force is Openness," *Chilton's I & CS*, Vol. 63, No. 4, Apr. 1990, pp. 41-45.

[19] S. Van Tyle, "Bringing Standards to Embedded Systems Design," *Electronic Design*, Vol. 44, No. 9, May 1996, pp. 37-38, 40-41.

[20] LP Electonik GmbH, "Requirements of Open, Modular Architecture Controllers for Applications in the Automotive Industry," Version 1.1, Dec. 13, 1994, http://www.lp-elektronik.com.

[21] C. A. Wiatrowski, "Using PC-DOS for Real-Time Control," *Chilton's I & CS,* Vol. 59, No. 8, Aug. 1986, pp. 51-54.

[22] R. M. Foard, "Multitasking Methods," *PC Tech J.,* Vol. 3, No. 3, Mar. 1986, pp. 49-61.

[23] R. H. Rehrig and E. J. Minger, "Prodding the PC To Perform Real-Time Multitasking," *Mech. Eng.,* Vol. 109, No. 10, pp. 53-55.

[24] R. Lampman and J. Franklin, "New Operating Systems, Standards Ease Control System Design and Use," *Chilton's I & CS,* Vol. 62, No. 2, Feb. 1989, pp. 41-47.

[25] S. Davis, "PC-Based Automation Provides Real-Time Performance," *Control Engineering,* Vol. 40, No. 3, Mid-Feb., 1993, pp. 8-12.

[26] F. A. Putnam, "A Firsthand Look at OS/2," *Chilton's I & CS,* Vol. 61, No. 10, Oct. 1988, pp. 57-60.

[27] J. Heaton, "A New Solution to Real-Time Plant-Floor Control," *Chilton's I & CS,* Vol. 61, No. 9, Sept. 1988, pp. 57-60.

[28] J. Heaton, "Microchannel: The Changing PC Architecture," *Chilton's I & CS,* Vol. 61, No. 9, Sept. 1988, pp. 57-60.

[29] J. Heaton, "Simplifying the Interfacing Task," *Chilton's I & CS,* Vol. 61, No. 12, Dec. 1988, pp. 35-38.

[30] J. Loo (Loo@luc.edu), "Papers on Real Time with Win95/NT?, " *Newsgroup comp.realtime, May 23, 1996.*

[31] DataPro Research, "An Overview of Microcomputer Operating Systems," *DataPro Reports on Microcomputers,* Vol. 3, 1989, pp. (CM55.005.0) 51-60.

[32] I. Singh, "UNIX and Real-Time: A Force-Fit That Works," *Chilton's I & CS,* Vol. 62, No. 2, Feb. 1989, pp. 49-56.

[33] C. T. Cole, "Real-Time UNIX: Fact or Fantasy?" *UNIX Review,* Vol. 8, No. 9, Sept. 1990, pp. 40-45.

[34] W. Barcikowski, M. Hawryluk, "Comparative Evaluation of SCO UNIX and QNX Operating Systems With Respect of Real-Time Issues," *Real-Time Systems '95 Proceedings of the 2nd Conference on Real-Time Systems,* Sept. 1995, pp. 191-203.

[35] T. Dayan (tal@netcom.com), "Papers on Real Time with Win95/NT?, *"Newsgroup comp.realtime, May 5, 1996.*

[36] T. Knutson (tomknutson@usa.pipeline.com), "Papers on Real Time with Win95/NT?," (See also info@pharlap.com) *Newsgroup comp.realtime, May 24, 1996.*

[37] S. Jones (stephen_jones@simphonics.com), "Real-Time Windows 95," (See also http://www.simphonics.com/realtime.htm) *Newsgroup comp.realtime, Jan. 1, 1996.*

[38] D. Lui, "Windows 95-Based CNC Controls," *Autofact 95 Conference Proceedings, Chicago IL, Nov. 13-16, 1995,* Dearborn, MI, Society of Manufacturing Engineers, 1995, Vol. 4, pp. 53-66.

[39] C. Watts, "Predictable I/O for Windows Systems," *Electronic Product Design,* Vol. 16, No. 11, Nov. 1995, pp. 51-52.

[40] H. Munz (LP.Office@t-online.de), "GUI and Real Time Control With Windows NT," (See also http://www.lp-elektronik.com) *Newsgroup comp.realtime, Nov. 14, 1996.*

[41] P. Cleveland, "Just What is EISA?" *Chilton's I & CS,* Vol. 62, No. 9, Sept. 1989, pp. 91-94.

[42] A. F. Harvey, "High Performance Data Acquisition: EISA Bus Opens PC Data-Transfer Bottlenecks," *Electronic Component News,* Vol. 10, No. 12, Dec. 1989, pp. 55-58.

[43] W. Labs, "PC Factions Queue Up for Control," *Chilton's I & CS,* Vol. 62, No. 4, Apr. 1989, pp. 37-40.

[44] S. Leibson, "PCI: Real Versus Ideal," *EDN (European Edition),* Vol. 40, No. 22, Oct. 26, 1995, pp. 59-62, 64, 66.

[45] J. Novellino, "Changing Technology Boost Data Acquisition," *Electronics Design,* Vol. 43, No. 13, June, 1995, pp. 141-142, 144.

[46] G. Koble, "PC-Based Control: Does it Start at the End of the Line?," *Control Engineering,* Vol. 43, No. 3, Mid-Feb. 1996, pp. 41-42, 44, 46-53.

[47] D. Shear, "Desktop DOS Goes Under Cover to Run Embedded Systems," *EDN (European Edition),* Vol. 39, No. 16, August 4, 1994, pp. 43-44, 46, 48.

[48] V. A. Price, ed., "Control System Design Part 3: The PC-Based System," *Intech,* Vol. 36, No. 7, July 1989, pp. 28-33.

[49] C. Yu, "Implementation of a Digital PID Controller in a Hierarchical Distributed Control System," Thesis, O.S.U.-Electrical and Computer Engineering, 1987.

[50] P. Matthews and B. Furht, "Evaluating Real-Time UNIX," *Embedded Systems Programming,* Vol. 4, No. 7, July 1991, pp. 28-36.

[51] C. M. Krishna and Y. H. Lee, "Real-Time Systems," *I.E.E.E. Computer,* Vol. 24, No. 5, May 1991, pp. 10-11.

[52] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach,* Morgan Kaufmann Publishers, Inc., San Mateo, CA, 1990.

[53] Hewlett Packard, "HP 1000 Specifications," *Test and Measurement Catalog,* 1982, 1986, and 1988.

[54] K. W. Christopher, B. A. Feigenbaum, and S. O. Saliga, *Developing Applications Using DOS,* John Wiley & Sons, Inc., New York, 1990.

[55] J. Prosise, "Tutor," *PC Magazine,* Vol. 9, No. 11, Aug. 1990, pp. 167-170.

[56] M. Hyman, *"Advanced DOS,"* MIS: Press, New York, 1988.

[57] J. Wright, "CE Marking Not Just a Rumor," *Compliance Engineering.,* Vol. 13, No. 2, 1996, pp. 9-10, 12.

[58] G. T. Kaplan, "What Are Your Control System Options?" *Chilton's I & CS,* Vol. 61, No. 6, June 1988, pp. 45-48.

[59] S. Srivastava, "Real-Time System Design Using the Personal Computer," Thesis, Univ. of Arizona-Department of Electrical and Computer Engineering, 1988.

[60] B. Zoeliner, "When is an Industrial Personal Computer a Must?" *Chilton's I & CS,* Vol. 61, No. 10, Oct. 1988, pp. 67-68.

[61] R. S. Lai, *Writing MS-DOS Device Drivers,* Addison-Wesley Publishing Co., Reading, MA, 1987.

[62] P. Norton, *Inside the IBM PC, Access to Advanced Features and Programming,* Brady Books, New York, NY, 1986.

[63] Byte-BOS Integrated Systems, "Multitasking Operating System Specification," San Francisco, CA.

[64] Micro Digital, Inc., "SMX, Simple Multitasking Executive Specification," Cypress, CA.

[65] P. G. Schreier, "Whether for Development or as a Target, Real-Time Environments take to PCs in Increasing Numbers," *Personal Engineering & Instrumentation News,* Vol. 8, No. 7, July 1991, pp. 25-32.

[66] T. Adams and R. Zimmerman, "An Analysis of the 8086 Instruction Set Usage in MS DOS Programs," *Proc. Third Symposium on Architectural Support for Programming Languages and Systems,* April 1989, pp. 152-161.

[67] Intel 386DX Microprocessor Programmer's Reference Manual, 1990, ISBN 1-55512-131-4

[68] Intel 386DX Microprocessor Hardware Reference Manual, 1990, ISBN 1-55512-111-X

[69] T. Yager, "Bringing Benchmarks Up to SPEC," *Byte Magazine,* Vol. 21, No. 3, March, 1996, pp. 145-146.

APPENDIX

APPENDIX

Test Platform Hardware Summary

	Platform 1	**Platform 2**	**Platform 3**	**Platform 4**
Processor:	Pentium	486DX	386DX	486SX
Clock speed:	75 MHz	50 MHz	20 MHz	33 MHz
Cache memory:	256 KB	256 KB	0 KB	256 KB
Motherboard:	Trenton Terminals	unknown	unknown	unknown
Memory:	16 MB	4 MB	2 MB	8 MB
BIOS:	AMI 10/10/94	AMI 11/11/92	AMI	AMI
System bus type:	ISA	VESA Local Bus	ISA	ISA
Video card:	Headland Technology	Intergra	Headland Technology	Trident
Video RAM:	256 KB	1 MB	256 KB	1 MB
Hard drive:	Maxtor 540 MB	Maxtor 340 MB	Seagate 20 MB	Maxtor 540 MB

Program Listings

Program listings are too long for print out in this document. The referenced programs are available from the author at 103063.47@CompuServe.com.